CHALMERS

Implementation of a Linux Workstation Based on The LEON Processor

Marcus Hellqvist

Master's Thesis
Electrical Engineering Program
CHALMERS UNIVERSITY OF TECHNOLOGY
Departement of Computer Science and Engineering
Division of Computer Engineering
Gothenburg 2005

All rights reserved. This publication is protected by law in accordance with “Lagen om Upphovsrätt, 1960:729”. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the authors. All products and names used in this thesis work are registered trademarks of their own respective corporation.

© Marcus Hellqvist, Gothenburg 2005

Abstract

The implementation of a Linux workstation based on the LEON processor is described in this report. The workstation is implemented on a FPGA board as a System on Chip (SOC) design with Linux as operating system. In the frame of this project have a PS/2 keyboard interface and a text-based VGA controller with AMBA interfaces been developed as IP cores in VHDL. Linux device drivers for kernel 2.0 and 2.6 have been written for these two IP cores.

A stand alone solution for the workstation is presented with the LEON3 SPARC V8 processor, the PS/2 IP core, the VGA IP core and additional cores from GRLIB IP library provided by Gaisler Research AB.

Sammanfattning

I denna rapport beskrivs implementation av en Linux arbetsstation baserad på LEON processorn. Arbetsstationen har implementerats som en System on Chip (SOC) design på ett FPGA-kort med Linux som operativsystem. Inom ramarna för projektet har två IP-block med AMBA interface utvecklats i VHDL, ett PS/2 interface för tangentbordshantering och en textbaserad VGA controller. Till dessa två IP-block har drivrutiner för Linux kernel 2.0 och 2.6 skrivits.

Rapporten beskriver implementationen av en fristående arbetsstation med Linux som operativsystem. Arbetsstationens består av processorn LEON SPARC V8, ett IP-block för PS/2 tangentbord, ett IP-block med en textbaserad VGA controller samt ytterligare IP-block från IP-biblioteket GRLIB, tillhandahållet av Gaisler Research AB.

Acknowledgement

I would like to thank my supervisor Jiri Gaisler for his support and giving me the opportunity to do my master's thesis at Gaisler Research AB.

Thanks to all of the other staff at Gaisler Research for their help and special thanks to Konrad Eisele for his support with the Linux device drivers.

Further on would I like to thank Lars Bengtsson at the department of Computer Engineering at Chalmers for undertaking this master's thesis.

Marcus Hellqvist
Gothenburg July, 2005

1 Introduction 1
2 Overview of the design environment 2
2.1 GRLIB 2
2.2 LEON3 2
2.3 AMBA 2
2.3.1 AHB on-chip bus 2
2.3.2 APB on-chip bus 3
2.4 The two-process model 5
2.5 The GR-XC3S-1500 LEON FPGA Development board 6
3 Design of a PS/2 keyboard with APB interface 7
3.1 General description 7
3.2 Communication 7
3.2.1 Device to host communication 8
3.2.2 Host to device communication 8
4 Design of a text based VGA controller with APB interface 10
4.1 General description 10
4.2 Video timing 10
4.3 Video out hardware 12
5 Video Memory 13
5.1 Dual port block RAM 13
5.2 Scrolling 14
6 Character ROM 16
6.1 Component declaration 16
7 Linux device drivers 17
8 Summary 19
9 References 20
Appendix A Scan codes 21
Appendix B Keyboard commands 23
Appendix C User's Manual 25
C.1 APBPS2 - PS/2 keyboard with APB interface 25
C.2 APBVGA - VGA controller with APB interface 30
Appendix D VHDL code 34
Appendix E Keymap table 46

1 Introduction

Gaisler Research developes and supports the LEON SPARC V8 processor, a synthesisable processor core for embedded applications. It is of interest to implement a LEON based Linux workstation to demonstrate the capabilities of the processor. This master's thesis project consist of developing a LEON based linux workstation. The workstation is based on the LEON3 processor with MMU and additional cores from the GRLIB IP (Intellectual Property) library developed by Gaisler Research. The complete workstation is realized as a System on Chip (SOC) design and implemented on a single Field Programmable Gate Array (FPGA). In the frame of this project a PS/2 keyboard interface and a text-based VGA controller have been developed in VHDL.

This master thesis have been done in three phases: definition, implementation and testing.

During the definition phase was the overall configuration of the system defined. Specifications for the PS/2 and VGA core were written.

The implementation phase consisted of two parts: hardware development and software integration. The VGA controller and the PS/2 interface were developed in VHDL and simulated using the Modelsim simulator. They were then implemented to a LEON multiprocessor design and the complete SOC design was simulated and verified in a board level VHDL model. After verification of the whole system it was implemented on a FPGA board, correct operation of the hardware was tested using the GRMON debug monitor developed by Gaisler Research. When the functionality had been validated, device drivers for the Linux operating system kernel 2.0 and 2.6.11 were written for the VGA controller and the PS/2 interface. The Linux operating system including the two new drivers were then installed on the FPGA board. A special version of Linux for embedded applications was available in form of Snapgear Embedded Linux [1].

In the testing phase was the system booted up with the two different Linux kernels.

This report describes the work with putting the workstation together, chapter two declare the design environment with a short presentation of the LEON3 processor and the bus architecture of the SOC design among others. Chapter three describes the functionality of the PS/2 IP core. The functionality of the VGA IP core can be found in chapter four and the video memory used by it in chapter five. Chapter six describes how a ROM containing the text font in use is used by the VGA controller. Finally is the Linux device drivers written for the PS/2 interface and the VGA controller described in chapter seven.

2 Overview of the design environment

2.1 GRLIB

The GRLIB IP-library is developed for SOC designs and is a set of reusable IP-cores [2]. The IP-cores are bus-centric around the AMBA AHB bus and uses a coherent method for simulation and synthesis. The library is vendor independent and expandable with support for different CAD-tools and target technologies. Using GRLIB gives the developer great possibilities to design his own SOC design. GRLIB is organized as a collection of VHDL libraries where each IP-vendor has its own library name. A library usually consists of a number of packages declaring the IP-cores as components and registers used by the core. A unique plug&play method gives the opportunity to configure and connect IP-cores to fit the designers demands. This without the need to change any global resources, ensuring that changes in one vendor's library don't affect other vendor's libraries. In the Gaisler Research IP-core's manual [3], LEON3 with additional IP-cores provided by Gaisler Research can be found.

Simulation and synthesis scripts are automatically generated by a global makefile and have compatibilities to the following simulation tools: Modelsim, NcSIM and GHDL. Synthesis tools from Synopsis, Synplify, Cadence, Altera and Xilinx are also supported.

2.2 LEON3

LEON3 is a 32-bit synthesisable processor core conformed to the SPARC V8 architecture and it is designed for embedded application by Gaisler Research. It uses a 7-stages pipeline and a separate instruction and data cache interface (Harvard architecture). The integer unit implements the full SPARC V8 standard including hardware divide and multiply instructions. LEON3 is the processor that the workstation within this project will be based on, it is a part of the GRLIB IP-library and a more detailed description of the processor can be find in [3]. LEON3 is designed using VHDL and is available under the GNU Public License (GPL).

2.3 AMBA

The *A*dvanced *M*icrocontroller *B*us *A*rchitecture specification [4] is an on-chip communication standard for designing high-performance embedded microcontrollers, developed by the ARM corporation. The IP cores in GRLIB is designed and centered around the on-chip AMBA-2.0 AHB/APB bus. It is chosen due to its market dominance (ARM processors), well documentation and because it can be used for free without license restrictions. Adding IP-cores to the AMBA buses is unfortunately not a straight forward procedure, additional "sideband" signals for automatic address decoding, interrupt steering and device identification have been added to the GRLIB. This to get support for "plug&play" capability and with that easier SOC designs.

2.3.1 AHB on-chip bus

The *A*dvanced *H*igh-performance *B*us supports multiple bus masters and is suitable for units with high-bandwidth operations. Following features are supported for achieving high-bandwidth:

- burst transfers
- split transactions
- single-cycle bus master handover
- single-clock edge operation
- non-tristate implementation
- wider data bus configurations (64/128 bits)

A typical AMBA AHB system design consist of four different components, AHB masters, AHB slaves, AHB arbiter and a AHB decoder.

There can be one or more AHB masters connected to the bus depending of the design. Only one AHB master at the time is allowed to work, bus masters are able to initiate read and write operations by providing address and control information.

AHB slaves responds to read and write operations within a specified address-space range. The slave then signals back to the active master if the operation was an success, failure or waiting of the data transfer. The most common AHB slaves are on-chip memory, off-chip memory interface and APB-bridges.

The AHB arbiter controls the bus so that only one master at the time is allowed do initiate bus transfers and only one arbiter is needed for the AHB bus. The AHB decoder is a central address decoder that provide a select signal for each slave on the bus.

The AHB bus is multiplexed since there are no tristate signals. In GRLIB input and output signals are grouped in VHDL records. Each unit has a separate record signal and the arbiter select which unit that is allowed to drive the shared bus from the record signals driven to the arbiter. Figure 1 shows the multiplexed AHB bus.

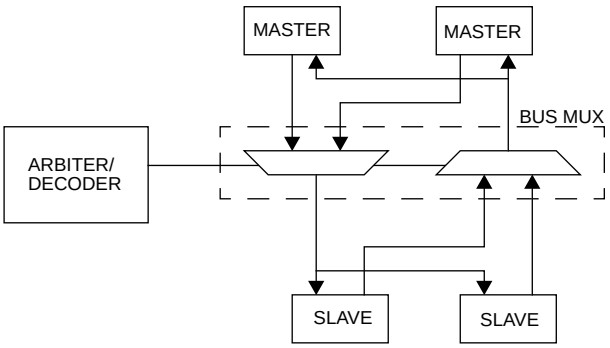

Figure 1. Multiplexed AMBA AHB

2.3.2 APB on-chip bus

The *A*dvanced *P*eripheral *B*us is a single-master bus optimized for minimal power consumption. The peripheral bus should be used for interface with low bandwidth and low complexity. The APB bus is connected to the AHB bus through a single AHB slave. This slave implements a AHB/APB bridge, it functions like a slave on the AHB bus and a master on the APB bus. The AHB/APB bridge is the only APB master on the specific bus, more than one APB bus can be connected to the AHB bus and therefore multiple AHB/APB bridges is supported. A conceptual view of the AMBA AHB/APB is viewed in figure 2.

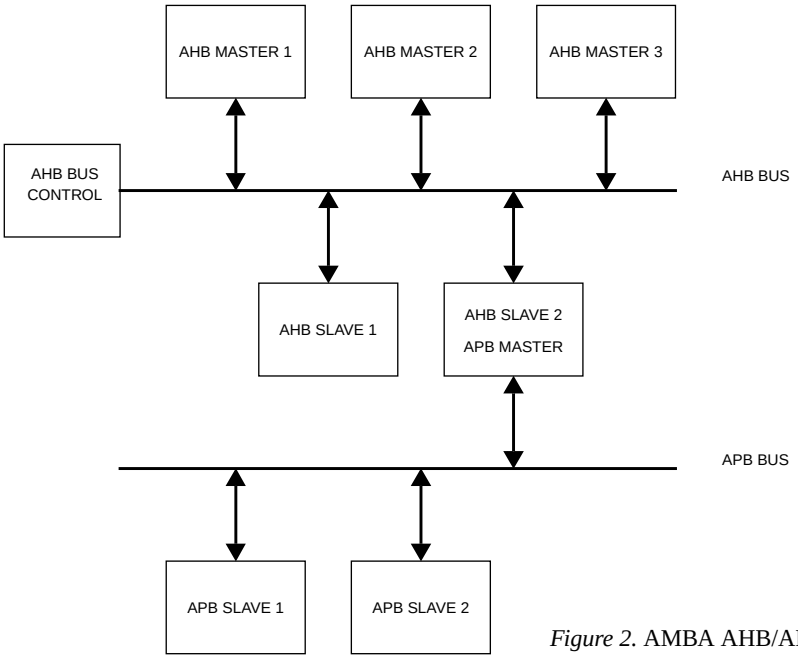

Figure 2. AMBA AHB/APB

The APB bus is multiplexed in similar way as the AHB bus. GRLIB provides a combined AHB slave, APB master, address decoder and bus multiplexer. The AHB/APB bridge receives the VHDL records AHBI and AHBO from the AHB bus and generates APBI and APBO on the APB bus figure 3 shows a multiplexed AMBA APB.

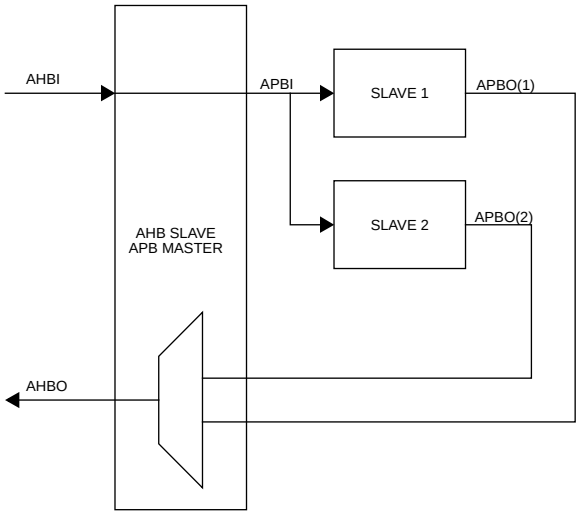

Figure 3. Multiplexed AMBA APB

The PS/2 interface and the VGA controller within this project are both interfaced with the APB bus as they are devices with low-bandwidth. A complete description of the signals and how to address the AHB and APB can be found in the GRLIB IP Library User's Manual [3].

2.4 The two-process model

The most common design style for synthesisable VHDL models is what we can call the dataflow style. It consists of a large number of concurrent VHDL statement and many small processes interconnected through signals building components with desired functionality. The dataflow style becomes difficult to understand and analyze when the number of concurrent statements increases. This because the concurrent statements and processes are not executed in the order they are written. The dataflow style arise from the old school hardware design where the engineers were used to schematic entry as design method. When they started to use VHDL as design tool the dataflow design style resembled of the schematics. The low complexity of the designs at that time, partly due to restrictions of the synthesis tools, made the dataflow style acceptable when coding VHDL. Today when the device complexity can reach millions of gates and the synthesis tools can handle a larger part of the VHDL standard a more efficient design method is needed.

Gaisler Research are using a newer different design style called the two-process model. A more detailed description and discussion of this model than the one described below can be found at [5]. Unlike the dataflow style which consists of a number of concurrent statements and processes the two-process model as the name reveal only uses two processes. One process that contains all combinational logic and one process that contains all sequential logic. With this structure can the algorithm be coded in sequential statements in the combinational process while the sequential process only contains registers. Sequential coding style are easier to follow when the execution of the statements are done from top to bottom, especially for larger designs.

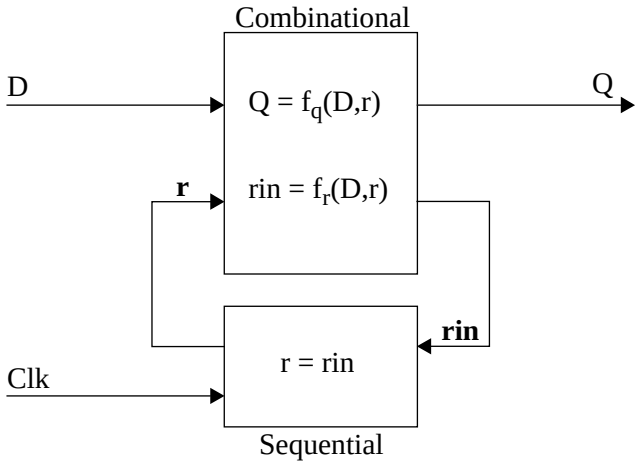

Figure 4. Two-process circuit

Figure 4 above shows a block diagram of the two process entity. Inputs to the entity are denoted D and connected to the combinational process. The sequential process have rin as inputs and r as outputs. rin are driven by the combinational process and are copied to r on the clock edge.

An additional thing to do for increasing the readability is to use record types to group associated signals, preferable is to group them according to functionality and direction. The record types can either be declared in a common global interface package or alternatively together with the component declaration in a component package. The package is then imported to those modules which uses the component. The benefit is that modifications in the design by removing or adding an element only needs to be done at one place, in the package where the record type is declared. Changes will then propagate through the whole design and time-consuming and error-prone manual editing is avoided.

By using record types for r and rin, elements can be added or removed in the record without any further modifications to the code. Neither the sensitivity list of the combinational process or the copying from rin to r in the sequential process needs to be modified. This because the operation is performed on the whole record irrespective of the number of elements in the record.

The PS/2 core and the VGA core designed within this master's thesis are both based on the two-process model. The VHDL code for these cores can be found in appendix D.

2.5 The GR-XC3S-1500 LEON FPGA Development board

The development board used for this project is a GR-XC3S-1500 developed by Pender Electronic Design in cooperation with Gaisler Research. It is a low cost development board targeted for the development of small LEON based systems, computer peripherals and as general purpose FPGA development environment. The board incorporates a 1.5 million gate XC3S1500 FPGA device from the Xilinx Spartan3 family. On-board memory in form of 8 mb FLASH prom and 64 mb SDRAM are provided together with ethernet, JTAG, serial, video, USB and PS/2 interfaces for off-board communication. This makes the board ideal for fast prototyping, evaluation and development of software for LEON microprocessor applications. A data sheet for the GR-XC3S can be found at [6]

For this project is the JTAG interface used for downloading of the design to the FPGA, the PS/2 interface for communication with the keyboard and the VGA connector together with an on-board 24-bit video DAC (ADV7125-50) used for communication with the monitor.

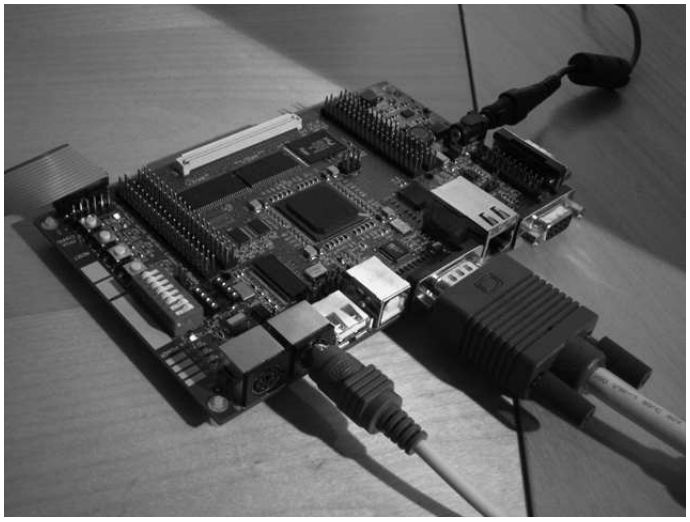

Figure 5. GR-XC3S-1500 development board

3 Design of a PS/2 keyboard with APB interface

3.1 General description

Keyboards consist of a large matrix of keys, each keypress and key release is detected by a built-in processor and encoded to scan codes. There are two kinds of scan codes, make codes for keys pressed and break codes for keys that are released. There are three standard sets of scan codes but today is the scan code set 2 the most common. In appendix A is a list of all possible scan codes for a 104-key keyboard, scan code set 2, available. Most of the keys use one byte for make codes and two bytes for break codes (e.g. ‘A’ has the make code “1C” and break code “F0 1C”). Some special keys have an extra prefix “E0” in both make and break codes (e.g. right control has make code “E0 14” and break code “E0 F0 14”).

Further in this section will the general communication for the PS/2 interface be described followed by the implementation of the interface to the AMBA APB bus.

3.2 Communication

The PS/2 keyboard use a bidirectional synchronous serial bus for transmitting and receiving data. The bus is idle when both the data and clock line are high. This is the only state where the keyboard is allowed to send data. Both the data line and the clock line are open-collectors with pull-up resistors to Vcc. If no data is sent on the bus, a resistor connected between the bus and the Vcc will pull the bus to a high-impedance state. Figure 6 shows a model of the electrical interface.

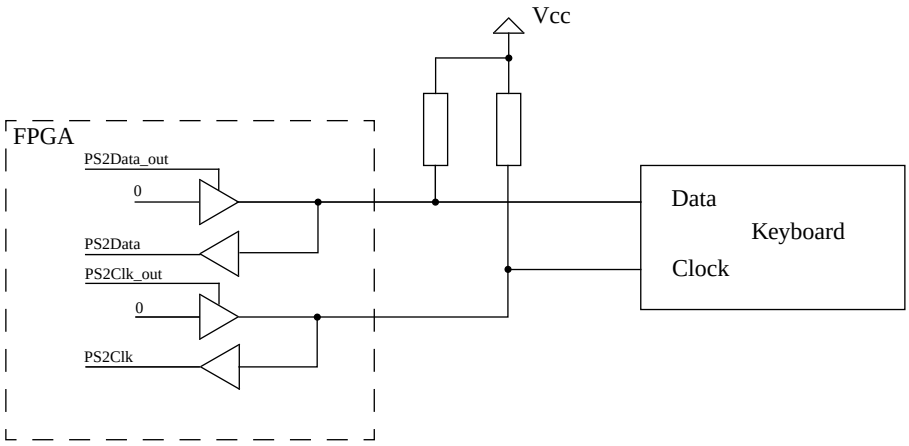

Figure 6. PS/2 electrical interface

The interface has full control of the communication and can easily inhibit the keyboard to transmit by pulling the clock line low. No data is sent until the host release the clock. While the communication is inhibited, the built-in controller in the keyboard buffer the data in a 16 byte buffer until the clock is released.

It is the keyboard controller which generate the clock and the clock frequency must be in the range 10-16.7 kHz. Still, the APB interface has full control over the communication and decides whether the keyboard controller is allowed or not to generate a clock signal.

The data is sent in a 11 bits serial frames, the first bit is a start bit followed by eight data bits, one odd parity bit and finally one stop bit. Figure 7 shows a PS/2 data frame.

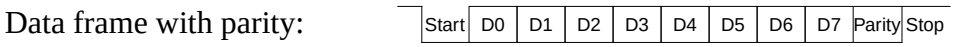

Figure 7. PS/2 data frame

The keyboard controller will from now on be referred as “device” in this document and the APB interface in the FPGA as “host”. There are two kinds of communication, device to host and host to device.

3.2.1 Device to host communication

When the keyboard wants to send data it first has to check if the clock line is in a high logic level. The clock line must continuously be high for at least 50 microseconds before the device can start to transmit. Then the device starts transmitting by pulling the data line low, sending the start bit which always is a logic zero. 5 to 25 microseconds after the data line is pulled low the clock starts to generate a signal with the frequency 10-16.7 kHz. Data sent from the device is read by the host on falling edge of the clock signal. After the start bit, eight data bits is sent with the least significant bit first. The following parity bit is set if there is an even number of ‘1’:s in the data frame and reset if there is an odd number of ‘1’:s. The finalizing stop bit is always a logic one.

The host may inhibit the transmission by pulling the clock line low for at least 100 microseconds. The device then has to retransmit the data. If the data is a make or break code consisting of more than one byte, the device must retransmit all bytes. If the clock is pulled low before the first falling edge in a transmission state or after the falling edge of the 11th clock cycle, the device must not retransmit the data.

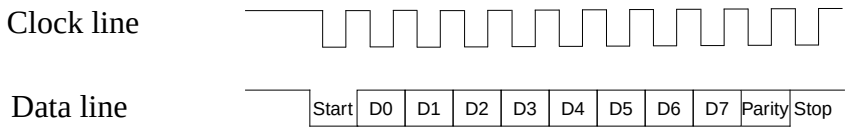

Figure 8. Device to host communication

3.2.2 Host to device communication

The host to device communication works a little bit different. Here the host has to request to send before transmission can be done. As before the host decides when any communication is allowed and simply pulls the clock line low for at least 100 microseconds. After that, the data line is also pulled low and then the clock is released (The device should look for the request-to-send state in intervals of at least 10 microseconds). The host will then wait for the device to bring the clock line low. When this happens the host starts to transmit the data which consists of eight data bits with the least significant bit first, one odd parity bit and one stop bit. In contrast to the device-to-host communication where the host clocked the data on falling edge, this transmission will be sent by the host on falling edge and read by the device on the rising edge of the clock signal. When the stop bit is received the device will send an acknowledgement by bringing the data line low and generate one last clock cycle. The host should release the data line after the last clock cycle or the device will continue to pulse the clock line and an error in transmission will occur.

As in the device-to-host communication the host may at any time inhibit or in this case abort the transmission by just pulling the clock line low for 100 microseconds. In figure 9 the transmission procedure is illustrated by dividing the host and device signal generation.

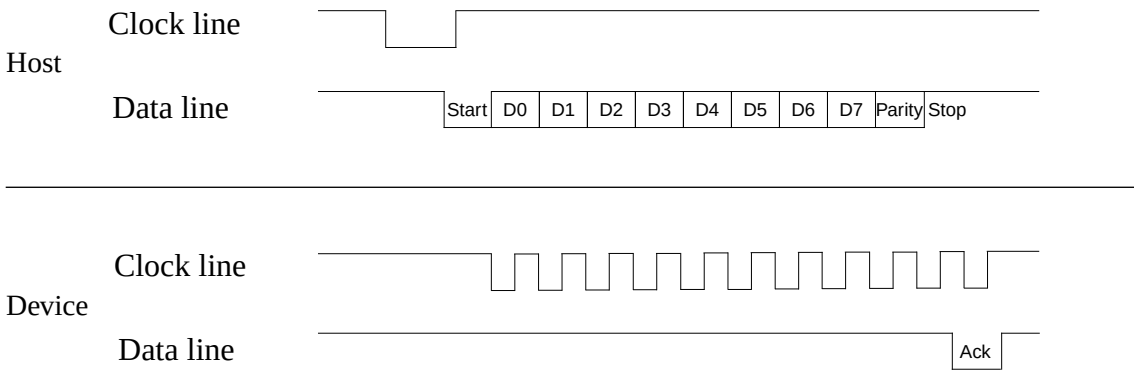

Figure 9. Detailed host to device communication

If the command sent by the host requires a response, the device must transmit that within 20 milliseconds after the host releases the clock line. If this does not happen the host should generate an error. Possible commands for the host to send is listed in appendix B.

A user's manual and the VHDL code for the PS/2 keyboard IP core written for this project can be found in appendix C.1 and appendix D.

4 Design of a text based VGA controller with APB interface

4.1 General description

The VGA controller designed for this project is text based and has the resolution 640x480 pixels. The controller is connected as a slave to the APB bus. As video buffer a 4 kb syncram_dp is used which is a dual port block RAM, a detailed description of this block RAM can be found in section 5. A character ROM which contains information about the font in use is also connected and a description of this memory is described in section 6. An address and a data frame is sent to the video controller via the VGA data register. The data frame is loaded into the video memory on the specific address, and are when ready to be used read by the video controller and decoded into bitmap fonts by the character ROM. As will be described in the chapter 6 each character correspond to a 8x13 pixel bitmap font. This means that a character is represented as thirteen bytes in the character ROM and every address in the video memory must be read thirteen times to display the character correct on the screen. A block diagram for the data path is shown in figure 10.

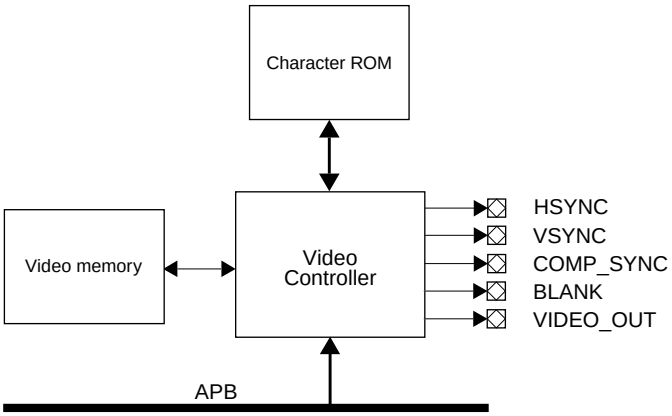

Figure 10. APB VGA block diagram

4.2 Video timing

Information about what and when something should be printed on the screen is sent to the monitor by the video controller. This is done by five different signals, RED, GREEN, BLUE, HSYNC and VSYNC. The three “colored” signals have information about the pixel data. Each color drive an electron gun that emits electrons to a point on the screen. All three signals have the size of 1 byte and combining different values gives different analog voltage levels to these control lines. A range of 0 V (completely darkness) to 0.7 V (maximum brightness) gives different intensities of the three primary colors to combine to put color to a pixel.

A single pixel or a line of pixels does not give much of information. A frame of 480 lines each one consisting of 640 pixels carries a bit more information and can present an image or a screen of text on the monitor. To print a frame the electron guns have to move from left to right, top to bottom. Movements in horizontal direction is controlled by the signal HSYNC, horizontal synchronization. Negative pulses on HSYNC define the start and the end of a line, this to ensure that the pixels are displayed at the visible part of the monitor. In similar fashion the negative pulses on the VSYNC, vertical synchronization, signal mark the top and bottom of the screen and ensures that all the 480 lines are displayed in the visible region of the monitor.

Information is only displayed in the forward direction, that is when the electron beam is moving left to right and top to bottom. When the beam is returning either to the left or to the top the electron beam is turned of, that is called the blanking region. Therefore much of the potential display time is lost. In figure 11 and 12 signal timing are shown for a 640 pixels by 480 lines display using a 25.175 MHz pixel clock and refresh rate of 60 Hz. The timing used in this design are based on the “VGA industry standard” 640x480 pixel mode 2 [7].

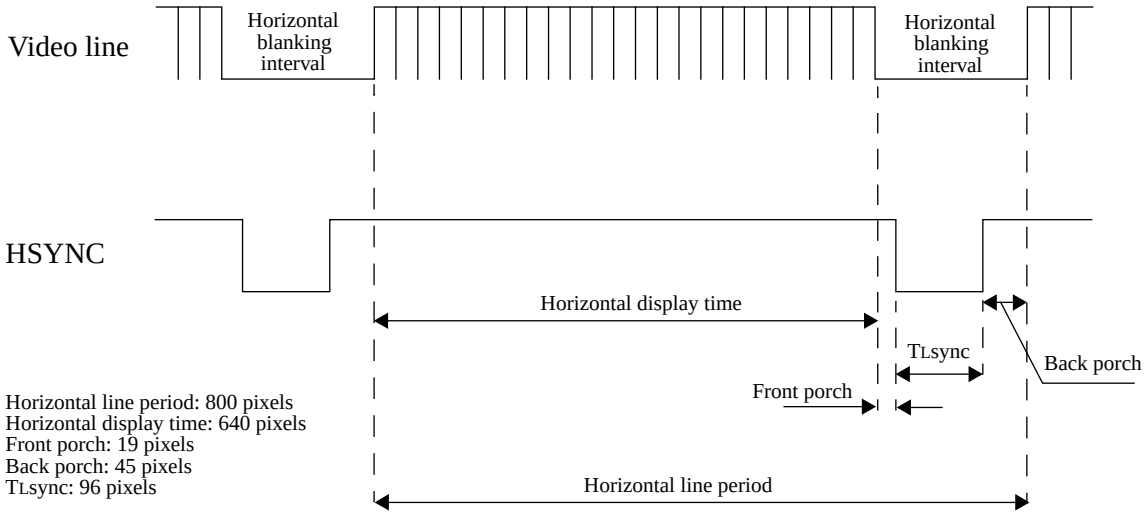

Figure 11. Horizontal synchronization timing

The horizontal display time is the time when video data is printed on the screen. The front porch of the sync pulse is the delay between the end of the video data and the initial falling edge of the sync pulse, TLsync. The back porch is the delay between the rising edge of the sync pulse and the first part of the video data referring to the next line. Each pixel is pulsed with a pixel clock, or dot clock with the frequency of 25.175 MHz.

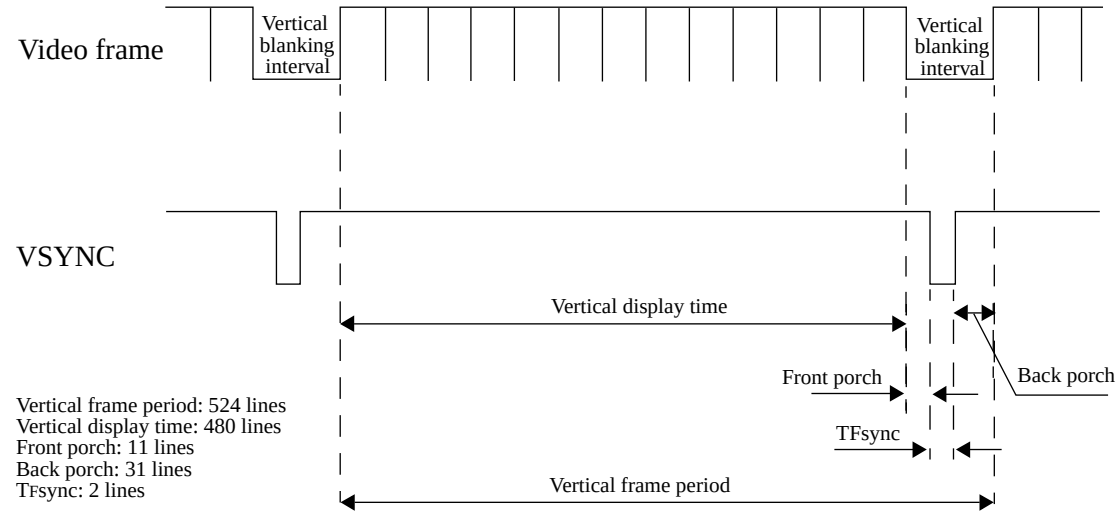

Figure 12. Vertical synchronization timing

The timing for vertical synchronization is analogues to the horizontal, the difference is that the timing unit is in lines instead of pixels. In figure 11 is the right border counted to the horizontal front porch and the left border to the horizontal back porch. Same counts for figure 12 where the bottom border is counted together with the vertical front porch and the top border with the vertical back porch. It does not affect anything, it is just two different ways to see it.

4.3 Video out hardware

To get visuals on the screen the digital color information has to be converted to analogue signals and this is done by a digital-to-analog converter (DAC). On the development board used for this project an Analog Devices AD7125 triple 8-bit DAC [8] handled the conversion. Figure 13 shows a block diagram of the data path in the ADV7125.

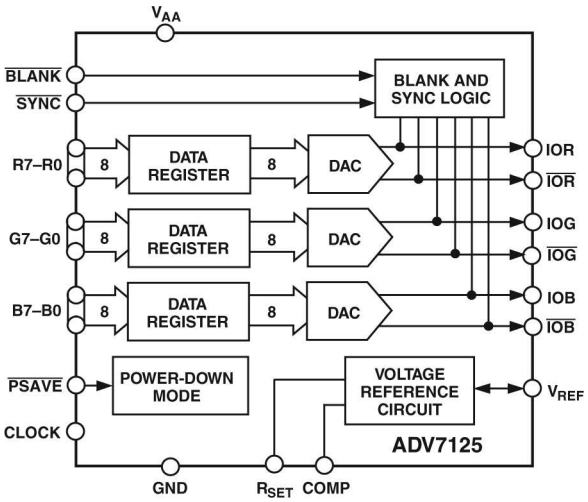

Figure 13. ADV7125 video DAC

The clock used by the ADV7125 is the pixel clock at 25.175 MHz. The RED, GREEN and BLUE 8-bit vectors are converted to analogue signals. $\overline{\text{BLANK}}$ and $\overline{\text{SYNC}}$ are control signals that controls that the pixel information is sent at the right time. By right time is meant when the electron guns works in the visible region of the screen, both signals are active low. $\overline{\text{BLANK}}$ correspond to the blanking signal which is set whenever HSYNC or VSYNC enters the blanking region. $\overline{\text{SYNC}}$ correspond to the composite sync signal which prevent that any pixel information is sent if either HSYNC or VSYNC is low. The signals on the right, IOR, IOG and IOB are connected to a standard VGA connector. HSYNC and VSYNC are connected directly from the FPGA.

$\overline{\text{SYNC}}$= not (HSYNC xor VSYNC)

$\overline{\text{BLANK}}$ = video_line and video_frame (see figure 11 and figure 12)

A user's manual and the VHDL code for the VGA controller IP core written for this project can be found in appendix C.2 and appendix D.

5 Video Memory

5.1 Dual port block RAM

The video memory is a 4 kb block RAM generated by the dual port RAM generator syncram_dp[3], provided by Gaisler Research. The reason that a syncram_dp is chosen is that it makes it possible to read and write to the memory at the same time and that it can be done with two different clock frequencies.

Data stored in the video memory is eight bit wide and correspond to an ascii value. Data is written to the memory when new data appears in the data register on the APB bus. Naturally the write mode is clocked by the system clock. Data is read from the memory every eighth clock pulse of the pixel clock. This because the data has to be “converted” to eight actual pixels in the character ROM look-up table and then sent to the monitor (see section 6). Figure 14 shows how the dual port RAM is connected.

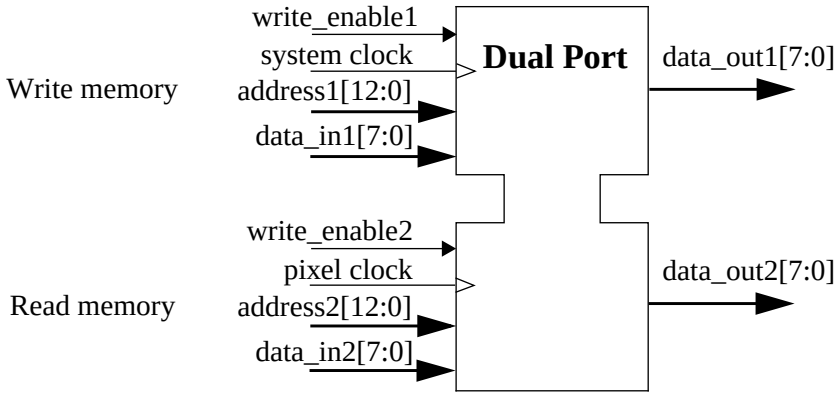

Figure 14. Dual port RAM

Writing to the memory is done on rising edge of the system clock and write_enable1 has to be set to ‘1’. data_out1 is never used because the data read is always shown on data_out2. Reading of the memory is done on the rising edge of the pixel clock, write_enable2 must always be set to ‘0’ to be in the reading mode.

5.2 Scrolling

The font in the character ROM uses 8 pixel by 13 rows per character. For a 640 pixel by 480 line display this means that 80 characters per line and 37 lines fits into the visible region. This correspond to 2960 bytes in the video memory. To get a steady image on the screen the video memory needs to loop these 2960 bytes over and over again. At startup does the 2960 bytes corresponds to the address range from 0x000 to 0xB8F in the video memory. The range of 0xB90 (2960 decimal) addresses will from now on be called Max_Frame. Figure 15 shows how the visible region of the screen correspond to the video memory.

Figure 15. Visible region of the screen

When the video controller exceeds writing the bottom right corner of the screen some kind of scroll function is needed. In hardware is this done by copying a reference address to a pointer which points to the next address to read from. This copying procedure is done when the video frame enters the vertical blanking region (see figure 12).

When the range between the reference address and the latest written address in the video memory exceeds the Max_Frame, the reference address is updated by 0x050. This corresponds to moving the reference address 80 places forward in the memory, in other words one line on the screen. By making this procedure, the visual text on the screen moves one line upwards. Figure 16 shows how the VGA controller scrolled the text one line upwards.

Figure 16. Screen with one scrolled line

The video memory is read and written by registers updated from two different processes in the VGA core, one “read process” which is clocked on the pixel clock and one “write process” which is clocked with the system clock. The reference address is updated by a register from the “write process” while the pointer for the last read address is updated by a register from the “read process”.

When copying the reference address to the pointer a problem with metastability may arise due to differences in the clock frequencies that they are updated on. To reduce the probability of metastability, the reference address is synchronized to the “read process” by running it through a flip-flop pulsed on the pixel clock before it is used.

Using two flip-flops in cascade is a better way to avoid metastability. Because that metastability is a statistical effect, the possibility of metastability diminishes with cascaded flip-flops [9]. But when the update of the pointer only happens when nothing is printed to the screen and the maximum update frequency of the reference address is every 80th clock cycle of the system clock, at the same time as the difference in frequency between the two clocks are not that high (about 25 MHz for the pixel clock and 50 MHz for the system clock), a second flip-flop in cascade is calculated to be unnecessary.

To be able to get the hardware scroller to work, the whole 4 kb video memory could not be used. This due to that 80 characters per line generates 51.2 lines out of a 4 kb memory. The 0.2 lines which correspond to 16 characters needed to be “cut of” in some how. Unfortunately could not a normal wrap around of the memory be used after writing to the last address in the memory. Instead will the video controller start to write to address 0x000 again after that the last address on line 51 is written. The same procedure is applied when the video controller is reading the memory.

In the Linux device drivers written for the VGA controller, an option to choose software scrolling instead of hardware scrolling is available. The software scroller only uses a part of the video memory corresponding to Max_Frame (with base address 0x000), in other words one screen. A function in the software scroller shift out the 80 least significant addresses which is equivalent to one line on the screen. In this way the bottom line on the screen gets empty and the text scroll up one line.

It is always the software which decides whether a hardware or software scroller shall be in use. The hardware will always scroll if data is written outside the Max_Frame, so it is important to not write outside this address range if a software scroller shall be in use.

6 Character ROM

When the video controller read an address from the video memory, it returns an eight bit data value. This value corresponds to a character and needs to be translated into pixels so that the monitor can display the character correctly. This translation is made with a character ROM which contains information of how every character is bit mapped into pixels. The ROM in use is an on-chip RAM that is used as read only stored with a bit mapped font. The font stored is based on the unicoded 8x13.bdf [10][11]. To not make the on-chip ROM to large only the characters with value below 256 are considered. Conveniently all the common ascii values are included among these.

The character font is bit mapped as 8 bits wide and 13 rows high. Each bit correspond to a pixel and a logic 1 means that the pixel should be colored with the foreground color and a 0 with the background color. Figure 17 shows how a capital ‘A’ is bit mapped.

Line	Value	
1	00	○ ○ ○ ○ ○ ○ ○ ○
2	00	○ ○ ○ ○ ○ ○ ○ ○
3	18	○ ○ ○ ● ● ○ ○ ○
4	24	○ ○ ● ○ ○ ● ○ ○
5	42	○ ● ○ ○ ○ ○ ● ○
6	42	○ ● ○ ○ ○ ○ ● ○
7	42	○ ● ○ ○ ○ ○ ● ○
8	7E	○ ● ● ● ● ● ● ○
9	42	○ ● ○ ○ ○ ○ ● ○
10	42	○ ● ○ ○ ○ ○ ● ○
11	42	○ ● ○ ○ ○ ○ ● ○
12	00	○ ○ ○ ○ ○ ○ ○ ○
13	00	○ ○ ○ ○ ○ ○ ○ ○

Figure 17. Bit mapped ‘A’

For a character to be displayed correctly a character font has to be read thirteen times, one time for each line. The VGA controller loops the part of the video memory that correspond to a scanline on the screen thirteen times to display the character correct. The character ROM is addressed by 12 bits, the four most significant tells what line of the character that should be read and the eight least significant what character to deal with. For example if the fifth line of an ‘A’ is demanded, the address corresponding is 0x541, where 41 is the ascii value for ‘A’. The data returned by the character ROM is 0x42 which then will be read by the VGA controller bit by bit. If the bit is set, the foreground color is sent to the DAC otherwise will the background color be sent. The character ROM is driven from the VGA controller with the pixel clock at 25.175 MHz.

6.1 Component declaration

```
library grlib;
use grlib.stdlib.all;

component charrom
port(
clk: in std_ulogic;
addr: in std_logic_vector(11 downto 0);
data : out std_logic_vector(7 downto 0)
);
end component;
```

7 Linux device drivers

After that the functionality of the PS/2 core and the VGA controller core had been validated, the Linux operating system was installed on the FPGA board. Linux for LEON3 is provided through a special version of the Snapgear Embedded Linux distribution and can be downloaded from [1]. This special distribution has support for two kernel versions. uClinux (linux-2.0.x) for non-MMU systems and linux-2.6.11 for MMU systems.

The original uClinux was a derivate of Linux 2.0 kernel intended for microcontrollers without MMUs. Today, uClinux includes Linux kernel releases for 2.0, 2.4 and 2.6 and supports designs with MMUs for some architectures. Support for the SPARC architecture, which LEON3 is based on, is not included among those and therefore linux kernel 2.6 is used for designs with MMUs in this project.

To be able to get the PS/2 keyboard and the VGA controller to work with Linux, device drivers needed to be written. For the linux 2.0 kernel a keyboard driver already existed in the keyboard.c file. However this driver was specially written for the Intel 8042 controller which is the most common controller for communications between keyboard and computer. A separate file with hardware specific parts like interrupt handling and initializing was written. From this file were functions in the keyboard.c called for the more generic scan code handling routines. The keyboard.c was partly rewritten and among other things was the port to read and write from changed so that it fitted the defined AMBA addresses.

One major difference between the Intel 8042 and the interface written within this project is that the Intel 8042 translates the scan codes sent by the microcontroller inside the keyboard, from scan code set 2 to scan code set 1. The reason of the translation is that the BIOS in the Intel 8042 only interpret scan code set 1. That kind of translation is not implemented in the PS/2 interface designed for this project when it occupies unnecessary logic. The major difference between set 1 and set 2 apart from that the make codes are different, is that the break codes for set 1 not begins with the prefix “F0”. Instead the break codes only consists of one byte (except for the special keys) and has a 80h larger value than the make code.

Scan codes sent by the PS/2 interface are translated by the keyboard drivers translation tables. These translation tables are generated by a program called *loadkeys* which is a standard Linux command. The program reads a file which contains keyboard table descriptions and then generate a translation table. The files with the keyboard table desriptions are called keymaps and must be written in a certain way. Each key has its own definition line and looks like below.

keycode keynumber = keysym keysym keysym...

The keynumber is the internal identification of the key, equivalent to the scan code of the key. Keysyms represent keyboard actions and these actions depends on what modifiers are in effect at that moment. There are eight different modifiers, each of these modifiers has an associated weight of power of two according to table 1

TABLE 1. Keysym modifiers and weight

Modifiers	Weight
Shift	1
AltGr	2
Control	4
Alt	8
ShiftL	16
ShiftR	32
CtrlL	64
CtrlR	128

Below is an example of how the key ‘9’ can be defined in two different ways.

keycode 70 = nine Shift keycode 70 = parenright AltGr keycode 70 = bracketright

keycode 70 = nine parenright bracketright

Keysyms can be given in decimal, octal, hexadecimal, unicode or symbolic notation. As we can see above “nine”, “parenright” and “bracketright” are symbolic notation and there are a number of these predefined symbolic notations. To increase readability and reduce typing-work and typing-errors you can give a map specific line that tells which modifiers to take in count.

In the Linux distribution a default keymap is included. This default keymap is however written for the scan code set 1 and a new one which supports scan code set 2 was written. This keymap which can be found in appendix E uses seven modifiers and they are defined at the top of the file by “keymaps 0-2,4-5,8,12”. By adding all modifiers in effect, the effective action of a key is found out. For instance control-alt gives keymaps 12 according to table 1. As can be seen above in the second example, the modifiers have been left out. This can be done when the modifiers are predefined like “keymaps 0-2,4-5,8,12”. A more detailed description of keymaps can be found in the Linux manual pages. The keymap in appendix E is written for scan code set 2 and is defined for a swedish keyboard.

The complexity of the driver for the VGA controller is lower than the driver for the PS/2 interface due to that the VGA controller does not use an interrupt function. The initializing process in Linux search for the VGA device through an AMBA scan function that returns the address for the VGA controller. The size of the memory needed to be defined and also the number of rows and columns for a full screen. The macro for writing to the screen was rewritten so that the drivers output were written to the address of the VGA controller.

For the Linux 2.6.11 kernel a complete keyboard driver already existed and it supported all three types of scan code sets. An input subsystem for the keyboard driver was written which initialized the keyboard, handled interrupt signals and output data. Unlike the driver for the 2.0 kernel where a new keymap was written, the default included keymap could be used this time. This because that the driver itself translates to scan code set 1 from the incoming scan code set 2. The keyboard driver for 2.6.11 supports english keyboard unlike the driver for 2.0 which supports swedish. Another difference is that the driver for 2.0 can light the leds on the keyboard, that is not implemented in the driver for 2.6.11. As for the 2.0 kernel, only the base address and the size of the video memory needed to be defined for the VGA driver in the 2.6.11 kernel.

8 Summary

The purpose of this master's thesis was to design a LEON based Linux workstation to show the capabilities of the LEON processor. Two IP cores written in VHDL have been developed in the frame of this project, a PS/2 keyboard interface and a text-based VGA controller. These two IP cores together with the LEON3 processor and additional IP cores from the GRLIB IP library, provided by Gaisler Research, were used as hardware for the workstation. The software consists of the Linux operating system kernel 2.0 and 2.6.11 which includes conformed device drivers for the PS/2 interface and the VGA controller, written in the frame of this master's thesis.

The two IP cores developed comply with the industrial standards. Synthesis of a Spartan3 FPGA showed that the PS/2 keyboard IP core occupies 472 LUTs on the FPGA and can be operated by designs with system clock frequency up to 99.4 MHz. The VGA controller IP core occupies 236 LUTs on the FPGA and can operated by designs with system clock frequency up to 81.1 MHz. A system clock with the frequency of 75 MHz should be enough for the most common FPGA designs. The PS/2 core was synthesized with 16 bytes reception and transmission FIFOs. It is worth mentioning that the number of LUTs occupied and the estimated clock frequency for the system clock may change considerably with smaller respectively larger FIFOs.

The complete workstation was finally tested by booting it up with the Linux operating system. Both the Linux kernel 2.0 and 2.6.11 was tested and the workstation was functioning as supposed to.

Various functions can be added to the VGA controller in the future, for instance might a movable cursor be useful. Multicolored and blinking text frames is another feature that could be implemented. Improvements in the software is surely possible and it is a must with increasing functionality of the VGA controller.

Finally could be said that a fully functioning design of a Linux workstation based on the LEON processor have been developed and a picture of the workstation can be found below.

Figure 18. Workstation with LEON processor

9 References

[1] Snapgear Linux for Leon. Available at URL: http://www.gaisler.com

[2] Gaisler J. A Dual-Use Open-Source VHDL IP Library. Proceedings of the MAPLD International Conference; 2004 Sep 8-10. Washington, D.C;2004.

[3] Gaisler J, Catovic E. Gaisler Research IP Core's Manual. 2005 Feb. Available at URL: http://www.gaisler.com

[4] AMBA specification Rev 2.0. 1999. Available at URL: http://www.gaisler.com/doc/amba.pdf

[5] Gaisler J. Fault Tolerant Microprocessors for Space applications. 41-50, Available at URL: http://www.gaisler.com/doc/vhdl2proc.pdf

[6] GR-XC3S, LEON Spartan-3 development board product sheet. Available at URL: http://www.pender.ch/docs/GR-XC3S_product_sheet.pdf

[7] VGA timing information. 2002; [1 screen]. Available at URL: http://www.epanorama.net/documents/pc/vga_timing.html

[8] ADV7125 - 330 MHz triple 8-bit high speed video DAC. 2002 Nov. Available at URL: http://www.analog.com/UploadedFiles/Data_Sheets/103728240ADV7125_0.pdf

[9] Xilinx XAPP077: Metastability considerations. 1997 Jan. Available at URL: http://www.coe.montana.edu/ee/courses/ee367/pdffiles/xapp077.pdf

[10] Unicode fonts and tools for x11. 2002 Nov. Available at URL: http://www.cl.cam.ac.uk/ mgk25/ucs-fonts.tar.gz

[11] UTF-8 and Unicode FAQ. 2005. Available at URL: http://www.cl.cam.ac.uk/ mgk25/unicode.html#utf-8

Appendix A Scan codes

TABLE 2. Scan code set 2, 104-key keyboard

KEY	MAKE	BREAK	--	KEY	MAKE	BREAK	--	KEY	MAKE	BREAK
A	1C	F0,1C		9	46	F0,46		[	54	FO,54
B	32	F0,32			`0E	F0,0E		INSERT	E0,70	E0,F0,70
C	21	F0,21		-	4E	F0,4E		HOME	E0,6C	E0,F0,6C
D	23	F0,23		=	55	FO,55		PG UP	E0,7D	E0,F0,7D
E	24	F0,24		\	5D	F0,5D		DELETE	E0,71	E0,F0,71
F	2B	F0,2B		BKSP	66	F0,66		END	E0,69	E0,F0,69
G	34	F0,34		SPACE	29	F0,29		PG DN	E0,7A	E0,F0,7A
H	33	F0,33		TAB	0D	F0,0D		U ARROW	E0,75	E0,F0,75
I	43	F0,43		CAPS	58	F0,58		L ARROW	E0,6B	E0,F0,6B
J	3B	F0,3B		L SHFT	12	FO,12		D ARROW	E0,72	E0,F0,72
K	42	F0,42		L CTRL	14	FO,14		R ARROW	E0,74	E0,F0,74
L	4B	F0,4B		L GUI	E0,1F	E0,F0,1F		NUM	77	F0,77
M	3A	F0,3A		L ALT	11	F0,11		KP /	E0,4A	E0,F0,4A
N	31	F0,31		R SHFT	59	F0,59		KP *	7C	F0,7C
O	44	F0,44		R CTRL	E0,14	E0,F0,14		KP -	7B	F0,7B
P	4D	F0,4D		R GUI	E0,27	E0,F0,27		KP +	79	F0,79
Q	15	F0,15		R ALT	E0,11	E0,F0,11		KP EN	E0,5A	E0,F0,5A
R	2D	F0,2D		APPS	E0,2F	E0,F0,2F		KP .	71	F0,71
S	1B	F0,1B		ENTER	5A	F0,5A		KP 0	70	F0,70
T	2C	F0,2C		ESC	76	F0,76		KP 1	69	F0,69
U	3C	F0,3C		F1	5	F0,05		KP 2	72	F0,72
V	2A	F0,2A		F2	6	F0,06		KP 3	7A	F0,7A
W	1D	F0,1D		F3	4	F0,04		KP 4	6B	F0,6B
X	22	F0,22		F4	0C	F0,0C		KP 5	73	F0,73
Y	35	F0,35		F5	3	F0,03		KP 6	74	F0,74
Z	1A	F0,1A		F6	0B	F0,0B		KP 7	6C	F0,6C
0	45	F0,45		F7	83	F0,83		KP 8	75	F0,75
1	16	F0,16		F8	0A	F0,0A		KP 9	7D	F0,7D
2	1E	F0,1E		F9	1	F0,01		]	5B	F0,5B
3	26	F0,26		F10	9	F0,09		;	4C	F0,4C
4	25	F0,25		F11	78	F0,78			52	F0,52
5	2E	F0,2E		F12	7	F0,07		,	41	F0,41
6	36	F0,36		PRNT SCRN	E0,12, E0,7C	E0,F0, 7C,E0, F0,12		.	49	F0,49
7	3D	F0,3D		SCROLL	7E	F0,7E		/	4A	F0,4A
8	3E	F0,3E		PAUSE	E1,14,77, E1,F0,14, F0,77	-NONE-				

TABLE 3. Windows multimedia scan codes

KEY	MAKE	BREAK
Next Track	E0, 4D	E0, F0, 4D
Previous Track	E0, 15	E0, F0, 15
Stop	E0, 3B	E0, F0, 3B
Play/Pause	E0, 34	E0, F0, 34
Mute	E0, 23	E0, F0, 23
Volume Up	E0, 32	E0, F0, 32
Volume Down	E0, 21	E0, F0, 21
Media Select	E0, 50	E0, F0, 50
E-Mail	E0, 48	E0, F0, 48
Calculator	E0, 2B	E0, F0, 2B
My Computer	E0, 40	E0, F0, 40
WWW Search	E0, 10	E0, F0, 10
WWW Home	E0, 3A	E0, F0, 3A
WWW Back	E0, 38	E0, F0, 38
WWW Forward	E0, 30	E0, F0, 30
WWW Stop	E0, 28	E0, F0, 28
WWW Refresh	E0, 20	E0, F0, 20
WWW Favorites	E0, 18	E0, F0, 18

TABLE 4. ACPI scan codes (Advanced Configuration and Power Interface)

KEY	MAKE	BREAK
Power	E0, 37	E0, F0, 37
Sleep	E0, 3F	E0, F0, 3F
Wake	E0, 5E	E0, F0, 5E

Appendix B Keyboard commands

TABLE 5. Host to device commands:

Command	Description
0xED	Set status LED's - keyboard will reply with ACK (0xFA). The host follows this command with an argument byte*
0xEE	Echo command - expects an echo response
0xF0	Set scan code set - keyboard will reply with ACK (0xFA) and wait for another byte. 0x01-0x03 which determines the scan code set to use. 0x00 returns the current set.
0xF2	Read ID - the keyboard responds by sending a two byte device ID of 0xAB 0x83
0xF3	Set typematic repeat rate - keyboard will reply with ACK (0xFA) and wait for another byte which determines the typematic rate.
0xF4	Keyboard enable - clears the keyboards output buffer, enables keyboard scanning and returns an acknowledgement.
0xF5	Keyboard disable - resets the keyboard, disables keyboard scanning and returns an acknowledgement.
0xF6	Set default - load default typematic rate/delay (10.9cps/500ms) and scan code set 2
0xFE	Resend - upon receipt of the resend command the keyboard will retransmit the last byte
0xFF	Reset - resets the keyboard

* bit 0 controls the scroll lock, bit 1 the num lock, bit 2 the caps lock, bit 3-7 are ignored

TABLE 6. Device to host commands:

Command	Description
0xFA	Acknowledge
0xAA	Power on self test passed (BAT completed)
0xEE	Echo respond
0xFE	Resend - upon receipt of the resend command the host should retransmit the last byte
0x00	Error or buffer overflow
0xFF	Error of buffer overflow

TABLE 7. The typematic rate/delay argument byte

MSB							LSB
0	DELAY	DELAY	RATE	RATE	RATE	RATE	RATE

TABLE 8. Typematic repeat rates

Bits 0-4	Rate (cps)		Bits 0-4	Rate (cps)		Bits 0-4	Rate (cps)		Bits 0-4	Rate (cps)
00h	30		08h	15		10h	7.5		18h	3.7
01h	26.7		09h	13.3		11h	6.7		19h	3.3
02h	24		0Ah	12		12h	6		1Ah	3
03h	21.8		0Bh	10.9		13h	5.5		1Bh	2.7
04h	20.7		0Ch	10		14h	5		1Ch	2.5
05h	18.5		0Dh	9.2		15h	4.6		1Dh	2.3
06h	17.1		0Eh	8.6		16h	4.3		1Eh	2.1
07h	16		0Fh	8		17h	4		1Fh	2

TABLE 9. Typematic delays

Bits 5-6	Delay (seconds)
00b	0.25
01b	0.5
10b	0.75
11b	1

Appendix C User's Manual

C.1 APBPS2 - PS/2 keyboard with APB interface

PS/2 interface operations

Receiver operation

The receiver is enabled for data reception through the receiver enable (RE) bit in the PS/2 control register. The RE control bit together with a high to low transition of the data line starts the state machine for receiving operations. The serial input that now arrives is shifted through an eight bit shift register on falling edge of the clock line. The eight data bits are XORed and the result is compared to the parity bit which is the ninth bit sent by the device in the data frame. If it does not match an error will occur. If a parity error or framing error occurs, the data frame will be thrown away. Otherwise the data will be transferred to a 16 byte FIFO and the data ready (DR) bit in the PS/2 status register will be set as long as the FIFO contains at least one data frame. When the FIFO is full, the output buffer full (OF) bit in the status register is set. The keyboard will be inhibited and buffer data until the FIFO gets read again. Interrupt is sent when a correct stop bit is received then it's up to the software to handle any resend operations if the parity bit is wrong. Figure 19 shows a flow chart for the operations of the receiver state machine.

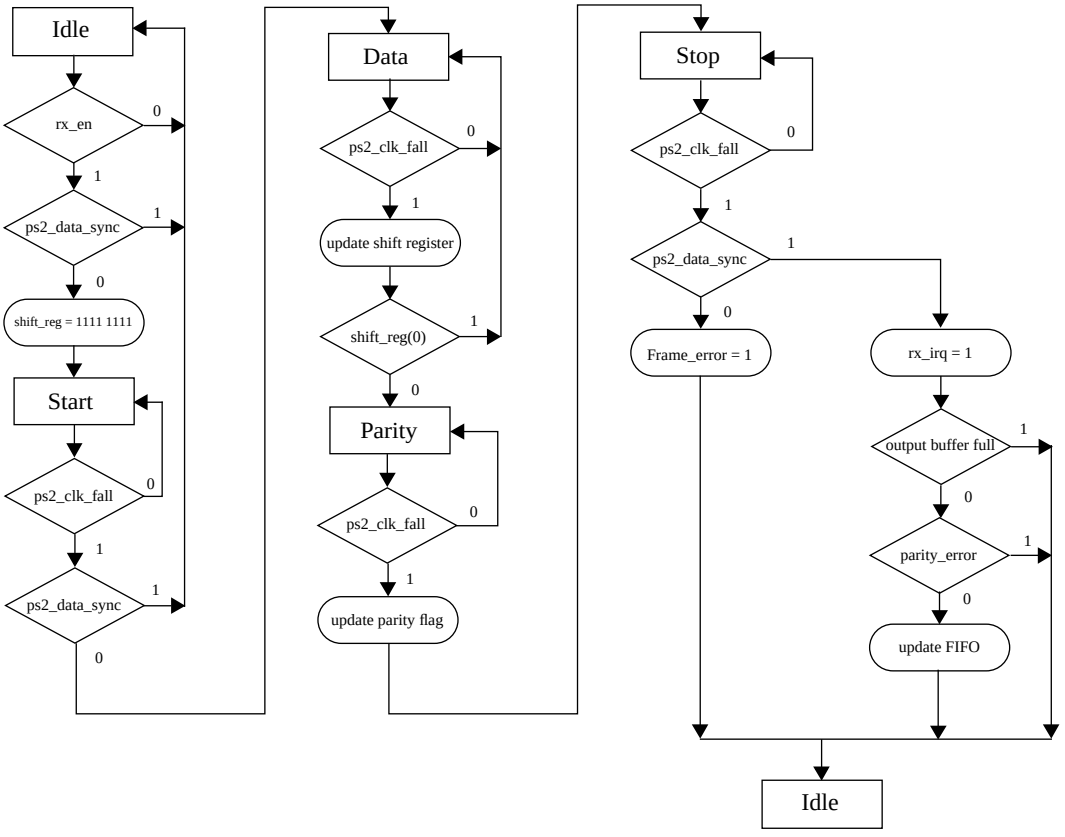

Figure 19. Flow chart for the receiver state machine

Transmit operations

The transmitter is enabled for data transmission through the transmitter enable (TE) bit in the PS/2 control register. The PS/2 interface has a 16 byte transmission FIFO that stores commands sent by the CPU. The most common commands are information about the leds on the keyboard and resending of the last sent byte, but also commands that sets the typematic rate and delay are available. Typematic rate is the repeat rate of a key that is held down, the typematic delay controls for how long a key has to be held down before it begins automatically repeating. Typematic repeat rates, delays and possible commands for the host to send are listed in appendix B.

If the TE bit is set and the transmission FIFO is not empty a transmission of the command will start. As described in section 3.2.2 the host will pull the clock line low for at least 100 us and then transmit a start bit, the eight bit command, an odd parity bit, a stop bit and wait for an acknowledgement bit by the device. When this happens an interrupt is sent and the host should be told to get into reception mode by the CPU. The reason that the host needs to be in the reception mode is that the device should have the opportunity to call for a resending of the last byte if invalid or acknowledge that the last byte was transmitted correctly. Not all commands are acknowledged by the device and some commands are followed by an argument byte. For instance the leds on the keyboard are set by a command byte 0xED followed by an argument byte with a value of 0 to 7. The three least significant bits correspond to the CAPS LOCK, NUM LOCK and SCROLL LOCK. When a command byte is followed by an argument byte, the device first acknowledge the command byte and then the argument byte. If the device calls for a resend, the host must resend both the command and argument byte. If the device is waiting for an argument byte and instead receives a command byte it should discard the previous command and process the new one. The device clears its output buffer when it receives any command from the host. Figure20 shows the flow chart for the transmission state machine.

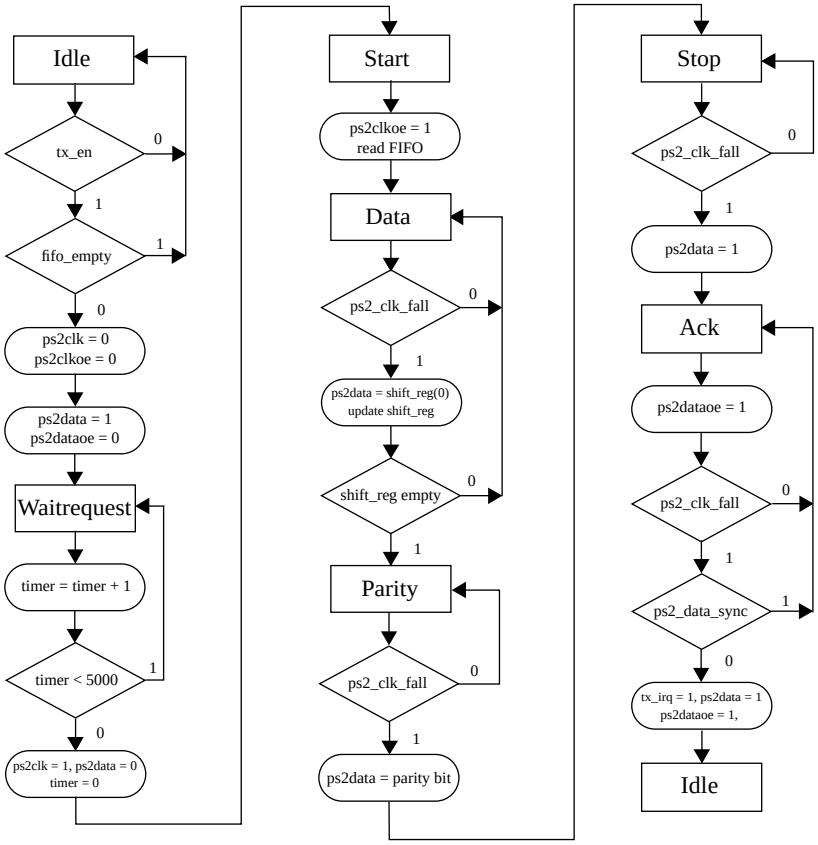

Figure 20. Flow chart for the transmitter state machine

The state machine for transmission needs to keep track of whether the host or the device shall use the clock and data line. This is done by two output enable signals, *ps2clkoe* and *ps2dataoe*, both active low. When an output enable signal is active the host set the value of the line and when it is inactive the device decides the value. This is necessary for instance when the device should start to generate the clock pulse and when it should acknowledge that the stop bit has arrived.

Configuration options

The APB PS/2 has the following configuration options (VHDL generics):

TABLE 10. APB PS/2 configuration options (VHDL generics)

Generic	Function	Allowed range	Default
pindex	APB slave index	0 - NAPBSLV-1	0
paddr	ADDR field of the APB BAR.	0 - 16#FFF#	0
pmask	MASK field of the APB BAR.	0 - 16#FFF#	16#FFF#
pirq	Index of the interrupt line.	0 - NAHBIRQ-1	0

Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x061. For a description of vendor and device ids see GRLIB IP Library User's Manual.

PS/2 registers

The APB PS/2 is controlled through three registers mapped into APB address space.

TABLE 11. APB PS/2 registers

Register	APB Address offset
PS/2 Data register	0x0
PS/2 Status register	0x4
PS/2 Control register	0x8

PS/2 Data Register

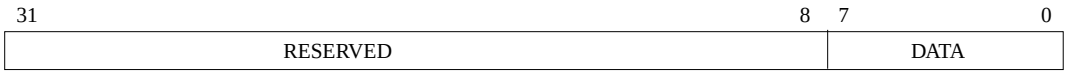

Figure 21. PS/2 data register

[7:0]: Receiver holding FIFO (read access)

PS/2 Status Register

Figure 22. PS/2 status register

0: Data ready (DR) - indicates that new data is available in the receiver holding register.
1: Parity error (PE) - indicates that a parity error was detected.
2: Framing error (FE) - indicates that a framing error was detected.
3: Keyboard inhibit (KI) - indicates that the keyboard is inhibited.
4: Output buffer full (OF) - indicates that the output buffer (FIFO) is full.
5: Input buffer full (IF) - indicates that the input buffer (FIFO) is full
[26:22]: Transmit FIFO count (TCNT) - shows the number of data frames in the transmit FIFO.
[31:27]: Receiver FIFO count (RCNT) - shows the number of data frames in the receiver FIFO.

PS/2 Control Register

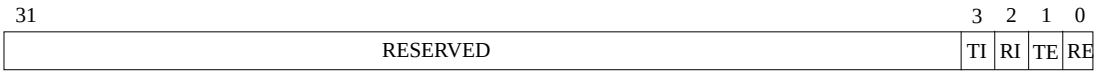

Figure 23. PS/2 control register

0: Receiver enable (RE) - if set, enables the receiver.
1: Transmitter enable (TE) - if set, enables the transmitter.
2: Keyboard interrupt enable (RI) - if set, interrupts are generated when a frame is received
3: Host interrupt enable (TI) - if set, interrupts are generated when a frame is transmitted

Signal descriptions

APB PS/2 signals are described in table 12.

TABLE 12. APB PS/2 signal descriptions.

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
PS2I	PS2_CLK_I	Input	PS/2 clock input	-
	PS2_DATA_I	Input	PS/2 data input	-
PS2O	PS2_CLK_O	Output	PS/2 clock output	-
	PS2_CLK_OE	Output	PS/2 clock output enable	Low
	PS2_DATA_O	Output	PS/2 data output	-
	PS2_DATA_OE	Output	PS/2 data output enable	Low

* see GRLIB IP Library User's Manual

Library dependencies

Table 13 shows libraries that should be used when instantiating an APB PS/2.

TABLE 13. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	APB signal definitions
GAISLER	PS2	Signals, component	PS/2 signal and component declaration

APB PS/2 instantiation

This example shows how to instantiate an APB PS/2

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library work;
use work.ps2.all;

entity apbps2_ex is
 port (
    rstn : in std_ulogic;
    clk : in std_ulogic;

    -- PS/2 signals
    ps2clk : inout std_ulogic;
    ps2data : inout std_ulogic
    );
end;

architecture rtl of apbuart_ex is

  -- APB signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);

  -- PS/2 signals
  signal ps2i : ps2_in_type;
  signal ps2o : ps2_out_type;

begin

  -- AMBA Components are instantiated here
   ...


  -- APB PS/2
  ps20 : apbps2
  generic map (pindex => 5, paddr => 5,  pirq => 4)
  port map (rstn, clk, apbi, apbo(5), ps2i, ps2o);


-- PS2 input/output data
  if ps2o.ps2_data_oe = '1' then
    ps2i.ps2_data_i <= ps2data;
  else
    ps2data <= ps2o.ps2_clk_o;
  end if;
```

```
      if ps2o.ps2_clk_oe = '1' then
        ps2i.ps2_clk_i <= ps2clk;
      else
        ps2clk <= ps2o.ps2_clk_o;
      end if;
  end;
```

C.2 APBVGA - VGA controller with APB interface

Operations

The VGA core contains three different registers, one data register, one register for the background color and one for the foreground color. Writing to the video memory is made through the VGA data register. The eight least significant bits contains information about what character that should be written. The twelve following bits corresponds to the address that should be written in the video memory. It is possible to set optional background and foreground color through the 24 least significant bits in the background and foreground registers. These 24 bits corresponds to the three pixel colors, RED, GREEN and BLUE. The eight most significant bits defines the red intensity, the next eight bits defines the green intensity and the eight least significant bits defines the blue intensity. Maximum intensity for a color is received when all eight bits are set and minimum intensity when none of the bits are set. Changing the foreground color results in that all characters change their color, it is not possible to just change the color of one character.

Configuration options

The APB VGA has the following configuration options (VHDL generics):

TABLE 14. APB VGA configuration options (VHDL generics)

Generic	Function	Allowed range	Default
memtech	Technology to implement on-chip RAM	0 - NTECH	2
pindex	APB slave index	0 - NAPBSLV-1	0
paddr	ADDR field of the APB BAR.	0 - 16#FFF#	0
pmask	MASK field of the APB BAR.	0 - 16#FFF#	16#FFF#

Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x060. For a description of vendor and device ids see GRLIB IP Library User's Manual.

VGA registers

The APB VGA is controlled through three registers mapped into APB address space.

TABLE 15. APB VGA registers

Register	APB Address offset
VGA Data register	0x0
VGA Background color	0x4
VGA Foreground color	0x8

VGA Data Register

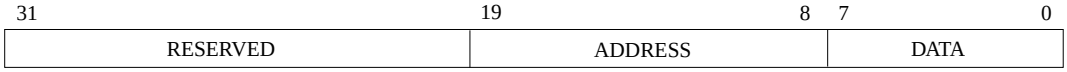

Figure 24. VGA data register

[19:8]: Video memory address (write access)
[7:0]: Video memory data (write access)

VGA Background Color

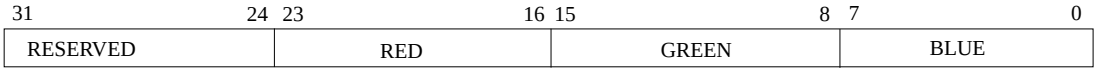

Figure 25. PS/2 status register

[23:16]: Video background color red.
[15:8]: Video background color green.
[7:0]: Video background color blue.

VGA Foreground Color

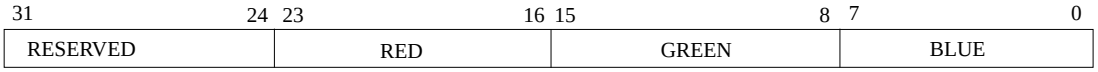

Figure 26. PS/2 status register

[23:16]: Video foreground color red.
[15:8]: Video foreground color green.
[7:0]: Video foreground color blue.

Signal descriptions

APB VGA signals are described in table 16.

TABLE 16. APB VGA signal descriptions.

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
VGACLK	N/A	Input	VGA Clock	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
HSYNC	N/A	Output	Horizontal synchronization	High
VSYNC	N/A	Output	Vertical synchronization	High
COMP_SYNC	N/A	Output	Composite synchronization	Low
BLANK	N/A	Output	Blanking	Low
VIDEO_OUT_R	N/A	Output	Video out, color red	-
VIDEO_OUT_G	N/A	Output	Video out, color green	-
VIDEO_OUT_B	N/A	Output	Video out, color blue	-

* see GRLIB IP Library User's Manual

Library dependencies

Table 17 shows libraries that should be used when instantiating an APB VGA.

TABLE 17. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	APB signal definitions
GAISLER	VGA	Signals, component	VGA signal and component declaration
GAISLER	MEMORY	Component	Component declaration
GAISLER	CHARROM_PACKAGE	Signals, component	Character ROM signal and component declaration

APB VGA instantiation

This example shows how to instantiate an APB VGA

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library work;
use work.vga.all;

entity apbvga_ex is
 port (
    rstn : in std_ulogic;
    clk : in std_ulogic;

    -- VGA signals
   vgaclk : in std_ulogic;
```

```
      hsync : out std_ulogic;
      vsync : out std_ulogic;
      csyncn : out std_ulogic;
      blankn : out std_ulogic;
      video_r : out std_logic_vector(7 downto 0);
      video_g : out std_logic_vector(7 downto 0);
      video_b : out std_logic_vector(7 downto 0
       );
  end;

  architecture rtl of apbuart_ex is

    -- APB signals
    signal apbi  : apb_slv_in_type;
    signal apbo  : apb_slv_out_vector := (others => apb_none);

  begin
    -- AMBA Components are instantiated here
     ...

    -- APB VGA
    vga0 : apbvga
    generic map (memtech => 2, pindex => 6, paddr => 6)
    port map (rstn, clk, vgaclk, apbi, apbo(6), hsync, vsync, csyncn,
              blankn, video_r, video_g, video_b);

end;
```

Appendix D VHDL code

```
------------------------------------------------------------------------------
-- Entity:      apbps2
-- File:        apbps2.vhd
-- Author:      Marcus Hellqvist
-- Description: PS/2 keyboard interface
------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

library grlib;
use grlib.stdlib.all;
use grlib.amba.all;
library gaisler;
use gaisler.devices.all;
library work;
use work.ps2.all;

entity apbps2 is
  generic(
    pindex      : integer := 0;
    paddr       : integer := 0;
    pmask       : integer := 16#fff#;
    pirq        : integer := 0
    );
  port(
    rst         : in std_ulogic;        -- Global asynchronous reset
    clk         : in std_ulogic;        -- Global clock

    apbi        : in apb_slv_in_type;
    apbo        : out apb_slv_out_type;

    ps2i        : in ps2_in_type;
    ps2o        : out ps2_out_type
    );
end entity apbps2;

architecture rtl of apbps2 is

constant fifosize       : integer := 16;
type rxstates is (idle,start,data,parity,stop);
type txstates is (idle,waitrequest,start,data,parity,stop,ack);
type fifotype is array(0 to fifosize-1) of std_logic_vector(7 downto 0);

type ps2_regs is record

-- status reg
 data_ready     : std_ulogic;                                   -- data ready
 parity_error   : std_ulogic;                                   -- parity carry out/ error bit
 frame_error    : std_ulogic;                                   -- frame error when receiving
 kb_inh         : std_ulogic;                                   -- keyboard inhibit
 obf            : std_ulogic;                                   -- output buffer full
 ibf            : std_ulogic;                                   -- input buffer full
 rcnt           : std_logic_vector(log2x(fifosize) downto 0);   -- fifo counter
 tcnt           : std_logic_vector(log2x(fifosize) downto 0);   -- fifo counter

-- control reg
 rx_en          : std_ulogic;                                   -- receive enable
 tx_en          : std_ulogic;                                   -- transmit enable
 rx_irq         : std_ulogic;                                   -- keyboard interrupt
 tx_irq         : std_ulogic;                                   -- transmit interrupt

-- others
```

```
 rxfifo         : fifotype;                                      -- fifo with 16 bytes
 rraddr         : std_logic_vector(log2x(fifosize)-1 downto 0); -- fifo read address
 rwaddr         : std_logic_vector(log2x(fifosize)-1 downto 0); -- fifo write address
 rxstate        : rxstates;
 txfifo         : fifotype;                                      -- fifo with 16 bytes
 traddr         : std_logic_vector(log2x(fifosize)-1 downto 0); -- fifo read address
 twaddr         : std_logic_vector(log2x(fifosize)-1 downto 0); -- fifo write address
 txstate        : txstates;
 ps2_clk_syn    : std_ulogic;                                    -- ps2 clock synchronized
 ps2_data_syn   : std_ulogic;                                    -- ps2 data synchronized
 ps2_clk_fall   : std_ulogic;                                    -- ps2 clock falling edge detect
 rshift         : std_logic_vector(7 downto 0);                  -- shift register
 rpar           : std_ulogic;                                    -- parity check bit
 tshift         : std_logic_vector(9 downto 0);                  -- shift register
 tpar           : std_ulogic;                                    -- transmit parity bit
 ps2clk         : std_ulogic;                                    -- ps2 clock
 ps2data        : std_ulogic;                                    -- ps2 data
 ps2clkoe       : std_ulogic;                                    -- ps2 clock output enable
 ps2dataoe      : std_ulogic;                                    -- ps2 data output enable
 timer          : std_logic_vector(12 downto 0);                 -- timer
end record;

 constant rcntzero      : std_logic_vector(log2x(fifosize) downto 0) := (others => '0');
 constant REVISION      : integer := 1;
 constant pconfig       : apb_config_type := (
                        0 => ahb_device_reg ( VENDOR_GAISLER, 16#61#, 0, REVISION, pirq),
                        1 => apb_iobar(paddr, pmask));

 signal r, rin          : ps2_regs;
 signal ps2_clk, ps2_data : std_ulogic;

begin
  ps2_op : process(r,rst,ps2_clk,ps2_data,apbi)
    variable v  : ps2_regs;
    variable rdata : std_logic_vector(31 downto 0);
    variable irq : std_logic_vector(15 downto 0);
  begin
    v := r;
    rdata := (others => '0'); v.data_ready := '0'; irq := (others => '0'); irq(pirq) := r.rx_irq
      or r.tx_irq;
    v.rx_irq := '0'; v.tx_irq := '0'; v.obf := r.rcnt(log2x(fifosize)); v.ibf :=
      r.tcnt(log2x(fifosize));

    if r.rcnt /= rcntzero then v.data_ready := '1'; end if;
    -- Synchronizing ps2 input
    v.ps2_clk_syn := ps2_clk;
    v.ps2_data_syn := ps2_data;

    -- read registers
    case apbi.paddr(3 downto 2) is
      when "00" =>      rdata(7 downto 0) := r.rxfifo(conv_integer(r.rraddr));
                        if (apbi.psel(pindex) and apbi.penable and (not apbi.pwrite)) = '1' then
                          if r.rcnt /= rcntzero then
                            v.rxfifo(conv_integer(r.rraddr)) := (others => '0');
                            v.rraddr := r.rraddr +1; v.rcnt := r.rcnt -1;
                          end if;
                        end if;
      when "01" =>      rdata(27 + log2x(fifosize) downto 27) := r.rcnt;
                        rdata(22 + log2x(fifosize) downto 22) := r.tcnt;
                        rdata(5 downto 0) := r.ibf & r.obf & r.kb_inh & r.frame_error &
      r.parity_error & r.data_ready;
      when "10" =>      rdata(3 downto 0) := r.tx_irq & r.rx_irq & r.tx_en & r.rx_en;
      when others =>
    end case;
```

```
-- write registers
if (apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1' then
  case apbi.paddr(3 downto 2) is
    when "00" =>    v.txfifo(conv_integer(r.twaddr)) := apbi.pwdata(7 downto 0);
                    v.twaddr := r.twaddr +1; v.tcnt := r.tcnt +1;
    when "01" =>    v.obf := apbi.pwdata(4);
                    v.kb_inh := apbi.pwdata(3);
                    v.frame_error := apbi.pwdata(2);
                    v.parity_error := apbi.pwdata(1);

    when "10" =>    v.tx_irq := apbi.pwdata(3);
                    v.rx_irq := apbi.pwdata(2);
                    v.tx_en := apbi.pwdata(1);
                    v.rx_en := apbi.pwdata(0);
    when others =>  null;
  end case;
end if;

if v.tx_en = '1' then
  v.rxstate := idle;
end if;

if r.ps2_clk_fall = '0' then
  if v.ps2_clk_syn /= r.ps2_clk_syn then
    if v.ps2_clk_syn = '0' and (v.rx_en = '1' or v.tx_en = '1') then
      v.ps2_clk_fall := '1';
    end if;
  end if;
else
  v.ps2_clk_fall := '0';
end if;

if v.tx_en = '0' then
-- receiver state machine
case r.rxstate is
when idle =>        if v.rx_en = '1' and v.ps2_data_syn = '0' then
                      v.rshift := (others => '1');
                      v.rxstate := start;
                    end if;
when start =>       if v.ps2_clk_fall = '1' then
                      if v.ps2_data_syn = '0' then
                        v.rshift := v.ps2_data_syn & r.rshift(7 downto 1);
                        v.rxstate := data;
                        v.rpar := '0';
                        v.parity_error := '0';
                        v.frame_error := '0';
                      else
                        v.rxstate := idle;
                      end if;
                    end if;
when data =>        if v.ps2_clk_fall = '1' then
                      v.rshift := v.ps2_data_syn & r.rshift(7 downto 1);
                      v.rpar := r.rpar xor v.ps2_data_syn;
                      if r.rshift(0) = '0' then
                        v.rxstate := parity;
                      end if;
                    end if;
when parity =>      if v.ps2_clk_fall = '1' then
                      v.parity_error := r.rpar xor (not v.ps2_data_syn);
                      v.rxstate := stop;
                    end if;
when stop =>        if v.ps2_clk_fall = '1' then
                      if v.ps2_data_syn = '1' then
                        v.rx_irq := '1';
                        v.rxstate := idle;
```

```
                        if (not v.obf) = '1' and r.parity_error = '0' then
                          v.rxfifo(conv_integer(r.rwaddr)) := v.rshift(7 downto 0);
                          v.rwaddr := r.rwaddr +1; v.rcnt :=r.rcnt +1;
                        end if;
                      else
                        v.frame_error := '1';
                        v.rxstate := idle;
                      end if;
                    end if;
end case;
end if;

-- keyboard inhibit / high impedance
if v.tx_en /= '1' then
  if r.obf = '1' then
    v.kb_inh := '1';
    v.ps2clk := '0' ;
    v.ps2data := '1';
    v.ps2dataoe := '0';
    v.ps2clkoe := '0';
  else
    v.ps2clk := '1';
    v.ps2data := '1';
    v.ps2dataoe := '1';
    v.ps2clkoe := '1';
  end if;
end if;

if r.tx_irq = '1' then
  v.tx_en := '0';
  v.rx_en := '1';
end if;

case r.txstate is
when idle =>        if v.tx_en = '1' and v.traddr /= v.twaddr then
                      v.ps2clk := '0';
                      v.ps2clkoe := '0';
                      v.ps2data := '1';
                      v.ps2dataoe := '0';
                      v.txstate := waitrequest;
                    end if;
when waitrequest => v.timer := r.timer +1;
                    if v.timer = conv_std_logic_vector(5000,13) then
                      v.timer := (others => '0');
                      v.ps2clk := '1';
                      v.ps2data := '0';
                      v.txstate := start;
                    end if;
when start  =>      v.ps2clkoe := '1';
                    v.tshift := "10" & r.txfifo(conv_integer(r.traddr));
                    v.traddr := r.traddr +1;v.tcnt := r.tcnt -1;
                    v.tpar := '1';
                    v.txstate := data;
when data =>        if v.ps2_clk_fall = '1' then
                      v.ps2data := r.tshift(0);
                      v.tpar := r.tpar xor r.tshift(0);
                      v.tshift := '1' & r.tshift(9 downto 1);
                      if v.tshift = "1111111110" then
                        v.txstate := parity;
                      end if;
                    end if;
when parity =>      if v.ps2_clk_fall = '1' then
                      v.ps2data := r.tpar;
                      v.txstate := stop;
                    end if;
```

```
    when stop =>        if v.ps2_clk_fall = '1' then
                          v.ps2data := '1';
                          v.txstate := ack;
                        end if;
    when ack =>         v.ps2dataoe := '1';
                        if v.ps2_clk_fall = '1' and v.ps2_data_syn = '0'then
                          v.ps2data := '1';
                          v.ps2dataoe := '0';
                          v.tx_irq := '1';
                          v.txstate := idle;
                        end if;
    end case;

    -- reset operations
    if rst = '0' then
      v.data_ready := '0';
      v.kb_inh := '0';
      v.parity_error := '0';
      v.frame_error := '0';
      v.obf := '0';
      v.rx_en := '0';
      v.tx_en := '0';
      v.rx_irq := '0';
      v.tx_irq := '0';
      v.ps2_clk_fall  := '0';
      v.ps2_clk_syn  := '0';
      v.ps2_data_syn := '0';
      v.rshift := (others => '0');
      v.rxstate := idle;
      v.txstate := idle;
      v.rraddr := (others => '0');
      v.rwaddr := (others => '0');
      v.rcnt := (others => '0');
      v.traddr := (others => '0');
      v.twaddr := (others => '0');
      v.tcnt := (others => '0');
      v.tshift := (others => '0');
      v.tpar := '0';
      v.timer := (others => '0');
    end if;

    -- update registers
    rin <= v;

    -- drive outputs
    apbo.prdata <= rdata; apbo.pirq <= irq;
    apbo.pindex <= pindex;
    ps2o.ps2_clk_o <= r.ps2clk;
    ps2o.ps2_clk_oe <= r.ps2clkoe;
    ps2o.ps2_data_o <= r.ps2data;
    ps2o.ps2_data_oe <= r.ps2dataoe;
  end process;

  apbo.pconfig <= pconfig;

  regs : process(clk)
  begin
    if rising_edge(clk) then
      r <= rin;
      ps2_data <= to_x01(ps2i.ps2_data_i);
      ps2_clk <= to_x01(ps2i.ps2_clk_i);
    end if;
  end process;

end architecture rtl;
```

```
------------------------------------------------------------------------------
-- Package:     ps2
-- File:        ps2.vhd
-- Author:      Marcus Hellqvist
-- Description: PS/2 types and component
------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

library grlib;
use grlib.stdlib.all;
use grlib.amba.all;
library gaisler;
use gaisler.devices.all;

package ps2 is

type ps2_in_type is record
 ps2_clk_i      : std_ulogic;
 ps2_data_i     : std_ulogic;
end record;

type ps2_out_type is record
 ps2_clk_o      : std_ulogic;
 ps2_clk_oe     : std_ulogic;
 ps2_data_o     : std_ulogic;
 ps2_data_oe    : std_ulogic;
end record;

component apbps2
  generic(
    pindex      : integer := 0;
    paddr       : integer := 0;
    pmask       : integer := 16#fff#;
    pirq        : integer := 0
    );
  port(
    rst         : in std_ulogic;                -- Global asynchronous reset
    clk         : in std_ulogic;                -- Global clock

    apbi        : in apb_slv_in_type;
    apbo        : out apb_slv_out_type;

    ps2i        : in ps2_in_type;
    ps2o        : out ps2_out_type

    );
end component apbps2;
end package;

------------------------------------------------------------------------------
-- Entity:      apbvga
-- File:        apbvga.vhd
-- Author:      Marcus Hellqvist
-- Description: VGA controller
------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
library grlib;
use grlib.stdlib.all;
use grlib.amba.all;
library gaisler;
use gaisler.devices.all;
use gaisler.memory.all;
use work.vga.all;
```

```
use work.charrom_package.all;

entity apbvga is
  generic(
    memtech     : integer := 2;
    pindex      : integer := 0;
    paddr       : integer := 0;
    pmask       : integer := 16#fff#
    );
  port( rst             : in std_ulogic;                        -- Global asynchronous reset
        clk             : in std_ulogic;                        -- Global clock
        vgaclk          : in std_ulogic;                        -- VGA clock

        apbi            : in apb_slv_in_type;
        apbo            : out apb_slv_out_type;

        hsync           : out std_ulogic;                       -- horizontal sync
        vsync           : out std_ulogic;                       -- vertical sync
        comp_sync       : out std_ulogic;                       -- composite sync
        blank           : out std_ulogic;                       -- blank signal

        video_out_r     : out std_logic_vector(7 downto 0);     -- pixel information
        video_out_g     : out std_logic_vector(7 downto 0);     -- pixel information
        video_out_b     : out std_logic_vector(7 downto 0)      -- pixel information
        );
end entity apbvga;

architecture rtl of apbvga is

type state_type is (s0,s1,s2);

 constant RAM_DEPTH     : integer := 12;
 constant RAM_DATA_BITS : integer := 8;
 constant MAX_FRAME     : std_logic_vector((RAM_DEPTH-1) downto 0):= X”B90”;

type ram_out_type is record
 dataout2               : std_logic_vector((RAM_DATA_BITS -1) downto 0);
end record;

type vga_regs is record
 video_out      : std_logic_vector(23 downto 0);
 hsync          : std_ulogic;
 vsync          : std_ulogic;
 csync          : std_ulogic;
 hcnt           : std_logic_vector(9 downto 0);
 vcnt           : std_logic_vector(9 downto 0);
 blank          : std_ulogic;
 linecnt        : std_logic_vector(3 downto 0);
 h_video_on     : std_ulogic;
 v_video_on     : std_ulogic;
 pixel          : std_ulogic;
 state          : state_type;
 rombit         : std_logic_vector(2 downto 0);
 romaddr        : std_logic_vector(11 downto 0);
 ramaddr2       : std_logic_vector((RAM_DEPTH -1) downto 0);
 ramdatain2     : std_logic_vector((RAM_DATA_BITS -1) downto 0);
 wstartaddr     : std_logic_vector((RAM_DEPTH-1) downto 0);
 raddr          : std_logic_vector((RAM_DEPTH-1) downto 0);
 tmp            : std_logic_vector(RAM_DEPTH-1 downto 0);
end record;

type color_reg_type is record
 bgcolor        : std_logic_vector(23 downto 0);
 txtcolor       : std_logic_vector(23 downto 0);
end record;
```

```
type vmmu_reg_type is record
 waddr          : std_logic_vector((RAM_DEPTH-1) downto 0);
 wstartaddr     : std_logic_vector((RAM_DEPTH-1) downto 0);
 ramaddr1       : std_logic_vector((RAM_DEPTH -1) downto 0);
 ramdatain1     : std_logic_vector((RAM_DATA_BITS -1) downto 0);
 ramenable1     : std_ulogic;
 ramwrite1      : std_ulogic;
 color          : color_reg_type;
end record;

constant REVISION       : amba_version_type := 0;
constant pconfig        : apb_config_type := (
                        0 => ahb_device_reg ( VENDOR_GAISLER, 16#60#, 0, REVISION, 0),
                        1 => apb_iobar(paddr, pmask));
constant hmax           : integer:= 799;
constant vmax           : integer:= 524;
constant hvideo         : integer:= 639;
constant vvideo         : integer:= 480;
constant hfporch        : integer:= 19;
constant vfporch        : integer:= 11;
constant hbporch        : integer:= 45;
constant vbporch        : integer:= 31;
constant hsyncpulse     : integer:= 96;
constant vsyncpulse     : integer:= 2;
constant char_height    : std_logic_vector(3 downto 0):=”1100”;

signal p,pin            : vmmu_reg_type;
signal ramo             : ram_out_type;
signal r,rin            : vga_regs;
signal romdata          : std_logic_vector(7 downto 0);

begin
  comb1: process(rst,r,p,romdata,ramo)
    variable v          : vga_regs;
    variable offset     : std_logic_vector(4 downto 0):=(others=>’0’);
  begin
    v:=r;
    v.wstartaddr := p.wstartaddr;

    -- horizontal counter
    if r.hcnt < conv_std_logic_vector(hmax,10) then
      v.hcnt := r.hcnt +1;
    else
      v.hcnt := (others => ‘0’);
    end if;
    -- vertical counter
    if (r.vcnt >= conv_std_logic_vector(vmax,10)) and (r.hcnt >= conv_std_logic_vector(hmax,10))
      then
      v.vcnt := (others => ‘0’);
    elsif r.hcnt = conv_std_logic_vector(hmax,10) then
      v.vcnt := r.vcnt +1;
    end if;
    -- horizontal pixel out
    if r.hcnt <= conv_std_logic_vector(hvideo,10) then
      v.h_video_on := ‘1’;
    else
      v.h_video_on := ‘0’;
    end if;
    -- vertical pixel out
    if r.vcnt <= conv_std_logic_vector(vvideo,10) then
      v.v_video_on := ‘1’;
    else
      v.v_video_on := ‘0’;
    end if;
```

```
-- generate hsync
if (r.hcnt <= conv_std_logic_vector((hvideo+hfporch+hsyncpulse),10)) and
  (r.hcnt >= conv_std_logic_vector((hvideo+hfporch),10)) then
  v.hsync := '0';
else
  v.hsync := '1';
end if;
-- generate vsync
if (r.vcnt <= conv_std_logic_vector((vvideo+vfporch+vsyncpulse),10)) and
  (r.vcnt >= conv_std_logic_vector((vvideo+vfporch),10)) then
  v.vsync := '0';
else
  v.vsync := '1';
end if;
--generate csync & blank
v.csync := not (v.hsync xor v.vsync);
v.blank := v.h_video_on and v.v_video_on;

-- count line of character
if v.hcnt = conv_std_logic_vector(hvideo,10) then
  if (r.linecnt = char_height) or (v.vcnt = conv_std_logic_vector(vmax,10)) then
    v.linecnt := (others => '0');
  else
    v.linecnt := r.linecnt +1;
  end if;
end if;

if v.blank = '1' then
  case r.state is
    when s0 =>      v.ramaddr2 := r.raddr;
                    v.raddr := r.raddr +1;
                    v.state := s1;
    when s1 =>      v.romaddr := v.linecnt & ramo.dataout2;
                    v.state := s2;
    when s2 =>      if r.rombit = "011" then
                      v.ramaddr2 := r.raddr;
                      v.raddr := r.raddr +1;
                    elsif r.rombit = "010" then
                      v.state := s1;
                    end if;
  end case;
  v.rombit := r.rombit - 1;
  v.pixel := romdata(conv_integer(r.rombit));
end if;

-- read from same address char_height times
if v.raddr = (r.tmp + X"050") then
  if (v.linecnt < char_height) then
    v.raddr := r.tmp;
  elsif v.raddr(11 downto 4) = X"FF" then       --check for end of allowed memory(80x51)
    v.raddr := (others => '0');
    v.tmp := (others => '0');
  else
    v.tmp := r.tmp + X"050";
  end if;
end if;

if  v.v_video_on = '0'      then
  v.raddr := r.wstartaddr;
  v.tmp := r.wstartaddr;
  v.state := s0;
end if;

-- define pixel color
if v.pixel = '1'and  v.blank = '1' then
```

```
    v.video_out := p.color.txtcolor;
  else
    v.video_out := p.color.bgcolor;
  end if;

  if rst = '0' then
    v.hcnt := conv_std_logic_Vector(hmax,10);
    v.vcnt := conv_std_logic_Vector(vmax,10);
    v.v_video_on := '0';
    v.h_video_on := '0';
    v.hsync := '0';
    v.vsync := '0';
    v.csync := '0';
    v.blank := '0';
    v.linecnt := (others => '0');
    v.state := s0;
    v.rombit := "111";
    v.pixel := '0';
    v.video_out := (others => '0');
    v.raddr := (others => '0');
    v.tmp := (others => '0');
    v.ramaddr2 := (others => '0');
    v.ramdatain2 := (others => '0');
  end if;

  -- update register
  rin <= v;

  -- drive outputs
  hsync <= r.hsync;
  vsync <= r.vsync;
  comp_sync <= r.csync;
  blank <= r.blank;
  video_out_r <= r.video_out(23 downto 16);
  video_out_g <= r.video_out(15 downto 8);
  video_out_b <= r.video_out(7 downto 0);
end process;


comb2: process(rst,r,p,apbi,ramo)
  variable v          : vmmu_reg_type;
  variable rdata : std_logic_vector(31 downto 0);
begin
  v := p;
  v.ramenable1 := '0'; v.ramwrite1 := '0';
  rdata := (others => '0');

  case apbi.paddr(3 downto 2) is
    when "00" =>      if (apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1' then
                         v.waddr := apbi.pwdata(19 downto 8);
                         v.ramdatain1 :=      apbi.pwdata(7 downto 0);
                         v.ramenable1 := '1';
                         v.ramwrite1  := '1';
                         v.ramaddr1 := apbi.pwdata(19 downto 8);
                       end if;
    when "01" =>      if (apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1' then
                        v.color.bgcolor := apbi.pwdata(23 downto 0);
                      end if;
    when "10" =>      if (apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1' then
                        v.color.txtcolor := apbi.pwdata(23 downto 0);
                      end if;
    when others =>    null;
  end case;

  if (p.waddr - p.wstartaddr) >= MAX_FRAME then
```

```
      if p.wstartaddr(11 downto 4) = X”FA” then    --last position of allowed memory
        v.wstartaddr := X”000”;
      else
        v.wstartaddr := p.wstartaddr + X”050”;
      end if;
    end if;

    if rst = ‘0’ then
      v.waddr    := (others => ‘0’);
      v.wstartaddr  := (others => ‘0’);
      v.color.bgcolor := (others => ‘0’);
      v.color.txtcolor := (others => ‘1’);
    end if;

    --update registers
    pin <= v;

    --drive outputs
    apbo.prdata <= rdata;
    apbo.pindex <= pindex;
    apbo.pirq <= (others => ‘0’);
  end process;

  apbo.pconfig <= pconfig;

  reg : process(clk)
  begin
    if clk’event and clk = ‘1’ then
      p <= pin;
    end if;
  end process;

  reg2 : process(vgaclk)
  begin
    if vgaclk’event and vgaclk = ‘1’ then
      r <= rin;
    end if;
  end process;

  rom0 : charrom port map(clk=>vgaclk, addr=>r.romaddr, data=>romdata);
  ram0 : syncram_dp generic map ( tech => memtech, abits => RAM_DEPTH, dbits => RAM_DATA_BITS)
    port map ( clk1 =>clk, address1=>p.ramaddr1, datain1=>p.ramdatain1, dataout1=>open,
               enable1=>p.ramenable1, write1=>p.ramwrite1, clk2=>vgaclk, address2=>r.ramaddr2,
               datain2=>r.ramdatain2, dataout2=>ramo.dataout2, enable2=>’0’,write2=> ‘0’);
end architecture;

-------------------------------------------------------------------------------
-- Package:     vhd
-- File:        vga.vhd
-- Author:      Marcus Hellqvist
-- Description: VGA component
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
library grlib;
use grlib.stdlib.all;
use grlib.amba.all;
library gaisler;
use gaisler.devices.all;
use gaisler.memory.all;

package vga is
component apbvga
  generic(
    memtech     : integer := 2;
```

```
    pindex      : integer := 0;
    paddr       : integer := 0;
    pmask       : integer := 16#fff#
    );
  port( rst             : in std_ulogic;                        -- Global asynchronous reset
        clk             : in std_ulogic;                        -- Global clock
        vgaclk          : in std_ulogic;                        -- VGA clock

        apbi            : in apb_slv_in_type;
        apbo            : out apb_slv_out_type;

        hsync           : out std_ulogic;                       -- horizontal sync
        vsync           : out std_ulogic;                       -- vertical sync
        comp_sync       : out std_ulogic;                       -- composite sync
        blank           : out std_ulogic;                       -- blank signal

        video_out_r     : out std_logic_vector(7 downto 0);     -- pixel information
        video_out_g     : out std_logic_vector(7 downto 0);     -- pixel information
        video_out_b     : out std_logic_vector(7 downto 0)      -- pixel information
        );
end component;
end package;
```

Appendix E Keymap table

```
# Default kernel keymap. This uses 7 modifier combinations.
keymaps 0-2,4-5,8,12
# Change the above line into
#keymaps 0-2,4-6,8,12
# in case you want the entries
#altgr   control keycode  83 = Boot
#altgr   control keycode 111 = Boot
# below.
#
# In fact AltGr is used very little, and one more keymap can
# be saved by mapping AltGr to Alt (and adapting a few entries):
# keycode 100 = Alt
#
keycode  1 = F9               F19             Console_21
control keycode  1 = F9
alt     keycode  1 = Console_9
control alt     keycode  1 = Console_9
keycode  3 = F5               F15             Console_17
control keycode  3 = F5
alt     keycode  3 = Console_5
control alt     keycode  3 = Console_5
keycode  4 = F3               F13             Console_15
control keycode  4 = F3
alt     keycode  4 = Console_3
control alt     keycode  4 = Console_3
keycode  5 = F1               F11             Console_13
control keycode  5 = F1
alt     keycode  5 = Console_1
control alt     keycode  5 = Console_1
keycode  6 = F2               F12             Console_14
control keycode  6 = F2
alt     keycode  6 = Console_2
control alt     keycode  6 = Console_2
keycode  7 = F12              F12             Console_24
control keycode  7 = F12
alt     keycode  7 = Console_12
control alt     keycode  7 = Console_12
keycode  9 = F10              F20             Console_22
control keycode  9 = F10
alt     keycode  9 = Console_10
control alt     keycode  9 = Console_10
keycode  10 = F8               F18             Console_20
control keycode  10 = F8
alt     keycode  10 = Console_8
control alt     keycode  10 = Console_8
keycode  11 = F6               F16             Console_18
control keycode  11 = F6
alt     keycode  11 = Console_6
control alt     keycode  11 = Console_6
keycode  12 = F4               F14             Console_16
control keycode  12 = F4
alt     keycode  12 = Console_4
control alt     keycode  12 = Console_4
keycode  13 = Tab              Tab
alt     keycode  13 = Meta_Tab
keycode  14 = sectiononehalf
keycode  15 =
keycode  16 =
keycode  17 = Alt
keycode  18 = Shift
keycode  19 =
keycode  20 = Control
```

```
keycode  21 = q
keycode  22 = one              exclam
alt     keycode   22 = Meta_one
keycode  23 =
keycode  24 =
keycode  25 =
keycode  26 = z
keycode  27 = s
keycode  28 = a
altgr   keycode  28 = Hex_A
keycode  29 = w
keycode  30 = two          quotedbl
controlkeycode   30 = nul
shiftcontrolkeycode   30 = nul
altkeycode   30 = Meta_two
altgrkeycode   30 = at
keycode  31 =
keycode  32 =
keycode  33 = c
control keycode  33 = Control_c
altgr   keycode  33 = Hex_C
keycode  34 = x
keycode  35 = d
altgr   keycode  35 = Hex_D
keycode  36 = e
altgr   keycode  36 = Hex_E
keycode  37 = four             currency           dollar
control keycode   37 = Control_backslash
alt     keycode   37 = Meta_four
keycode  38 = three            numbersign
control keycode   38 = Escape
alt     keycode   38 = Meta_three
altgr   keycode  38 = sterling
keycode  39 =
keycode  40 =
keycode  41 = space            space
control keycode  41 = nul
alt     keycode  41 = Meta_space
keycode  42 = v
keycode  43 = f
altgr   keycode  43 = Hex_F
keycode  44 = t
keycode  45 = r
keycode  46 = five             percent
control keycode   46 = Control_bracketright
alt     keycode   46 = Hex_E
keycode  47 =
keycode  48 =
keycode  49 = n
keycode  50 = b
altgr   keycode  50 = Hex_B
keycode  51 = h
keycode  52 = g
keycode  53 = y
keycode   54 = six              ampersand
control keycode   54 = Control_asciicircum
alt     keycode   54 = Meta_six
keycode  55 =
keycode  56 =
keycode  57 =
keycode  58 = m
keycode  59 = j
keycode  60 = u
keycode   61 = seven            slashbraceleft
control keycode   61 = Control_underscore
```

```
alt     keycode   61 = Meta_seven
keycode   62 = eight            parenleft         bracketleft
control keycode   62 = Delete
alt     keycode   62 = Meta_eight
keycode  63 =
keycode  64 =
keycode  65 = comma        semicolon
alt     keycode  65 = Meta_comma
keycode  66 = k
keycode  67 = i
keycode  68 = o
keycode  69 = zero            equal       braceright
alt     keycode  69 = Meta_zero
keycode  70 = nine             parenright        bracketright
alt     keycode  70 = Meta_nine
keycode  71 =
keycode  72 =
keycode  73 = period       colon
control keycode  73 = Compose
alt     keycode  73 = Meta_period
keycode  74 =minus            underscore
control keycode  74 = Delete
alt     keycode  74 = Meta_slash
keycode  75 = l
keycode  76 = +odiaeresis Odiaeresis
alt     keycode  76 = Meta_semicolon
keycode  77 = p
keycode  78 = plusquestion
altgr keycode 78 = backslash
controlkeycode  78 = Control_underscore
shiftcontrolkeycode  78 = Control_underscore
altkeycode  78 = Meta_minus
keycode  79 =
keycode  80 =
keycode  81 =
keycode  82 = +adiaeresisAdiaeresis
control keycode  82 = Control_g
alt     keycode  82 = Meta_apostrophe
keycode  83 =
keycode  84 = +aring Aring
control keycode  84 = Escape
alt     keycode  84 = Meta_bracketleft
keycode  85 =  dead_acute dead_grave
keycode  86 =
keycode  87 =
keycode  88 = Caps_Lock
keycode  89 = Shift
keycode  90 =  Return
keycode  91 = dead_diaeresis dead_circumflex dead_tilde
keycode  92 =
keycode  93 = apostropheasterisk
control keycode  93 = Control_backslash
alt     keycode  93 = Meta_backslash
keycode  94 =
keycode  95 = Find
keycode  96 = Pause
keycode  97 = lessgreater       bar
keycode  98 =  Delete
control keycode 98 = Control_w
keycode  99 =  Left
        alt     keycode 105 = Decr_Console
keycode 100 = AltGr
keycode 101 = Break
keycode 102 = Delete
keycode 103 = Up
```

```
keycode 104 = Prior
keycode  105 =KP_1
alt     keycode  105 = Ascii_1
altgr   keycode  105 = Hex_1
keycode 106 = Right
alt     keycode 106 = Incr_Console
keycode  107 = KP_4
alt     keycode  107 = Ascii_4
altgr   keycode  107 = Hex_4
keycode  108= KP_7
alt     keycode  108 = Ascii_7
altgr   keycode  108 = Hex_7
keycode 109 =
keycode 110 = Insert
keycode 111 =
keycode  112 = KP_0
alt     keycode  112 = Ascii_0
altgr   keycode  112 = Hex_0
keycode  113 = KP_Period
control alt     keycode  113 = Boot
keycode  114 = KP_2
alt     keycode  114 = Ascii_2
altgr   keycode  114 = Hex_2
keycode  115 = KP_5
alt     keycode  115 = Ascii_5
altgr   keycode  115 = Hex_5
keycode  116 = KP_6
alt     keycode  116 = Ascii_6
altgr   keycode  116 = Hex_6
keycode  117 = KP_8
alt     keycode  117 = Ascii_8
altgr   keycode  117 = Hex_8
keycode   118 = Escape           Escape
alt     keycode   118 = Meta_Escape
keycode  119 = Num_Lock
shift   keycode  119 = Bare_Num_Lock
keycode  120 = F11              F11              Console_23
control keycode  120 = F11
alt     keycode  120 = Console_11
control alt     keycode  120 = Console_11
keycode  121 = KP_Add
keycode  122 = KP_3
alt     keycode  122 = Ascii_3
altgr   keycode  122 = Hex_3
keycode  123 = KP_Subtract
keycode  124 = KP_Multiply
keycode  125 = KP_9
alt     keycode  125 = Ascii_9
altgr   keycode  125 = Hex_9
keycode  126 = Scroll_Lock      Show_Memory      Show_Registers
control keycode  126 = Show_State
alt     keycode  126 = Scroll_Lock
keycode 127 = slash
string F1 = “\033[[A”
string F2 = “\033[[B”
string F3 = “\033[[C”
string F4 = “\033[[D”
string F5 = “\033[[E”
string F6 = “\033[17~”
string F7 = “\033[18~”
string F8 = “\033[19~”
string F9 = “\033[20~”
string F10 = “\033[21~”
string F11 = “\033[23~”
string F12 = “\033[24~”
```

```
compose ‘`’ ‘A’ to ‘À’
compose ‘`’ ‘a’ to ‘à’
compose ‘\’’ ‘A’ to ‘Á’
compose ‘\’’ ‘a’ to ‘á’
compose ‘^’ ‘A’ to ‘Â’
compose ‘^’ ‘a’ to ‘â’
compose ‘~’ ‘A’ to ‘Ã’
compose ‘~’ ‘a’ to ‘ã’
compose ‘”’ ‘A’ to ‘Ä’
compose ‘”’ ‘a’ to ‘ä’
compose ‘O’ ‘A’ to ‘Å’
compose ‘o’ ‘a’ to ‘å’
compose ‘0’ ‘A’ to ‘Å’
compose ‘0’ ‘a’ to ‘å’
compose ‘A’ ‘A’ to ‘Å’
compose ‘a’ ‘a’ to ‘å’
compose ‘A’ ‘E’ to ‘Æ’
compose ‘a’ ‘e’ to ‘æ’
compose ‘,’ ‘C’ to ‘Ç’
compose ‘,’ ‘c’ to ‘ç’
compose ‘`’ ‘E’ to ‘È’
compose ‘`’ ‘e’ to ‘è’
compose ‘\’’ ‘E’ to ‘É’
compose ‘\’’ ‘e’ to ‘é’
compose ‘^’ ‘E’ to ‘Ê’
compose ‘^’ ‘e’ to ‘ê’
compose ‘”’ ‘E’ to ‘Ë’
compose ‘”’ ‘e’ to ‘ë’
compose ‘`’ ‘I’ to ‘Ì’
compose ‘`’ ‘i’ to ‘ì’
compose ‘\’’ ‘I’ to ‘Í’
compose ‘\’’ ‘i’ to ‘í’
compose ‘^’ ‘I’ to ‘Î’
compose ‘^’ ‘i’ to ‘î’
compose ‘”’ ‘I’ to ‘Ï’
compose ‘”’ ‘i’ to ‘ï’
compose ‘-’ ‘D’ to ‘D’
compose ‘-’ ‘d’ to ‘d’
compose ‘~’ ‘N’ to ‘Ñ’
compose ‘~’ ‘n’ to ‘ñ’
compose ‘`’ ‘O’ to ‘Ò‘
compose ‘`’ ‘o’ to ‘ò’
compose ‘\’’ ‘O’ to ‘Ó’
compose ‘\’’ ‘o’ to ‘ó’
compose ‘^’ ‘O’ to ‘Ô’
compose ‘^’ ‘o’ to ‘ô’
compose ‘~’ ‘O’ to ‘Õ’
compose ‘~’ ‘o’ to ‘õ’
compose ‘”’ ‘O’ to ‘Ö’
compose ‘”’ ‘o’ to ‘ö’
compose ‘/’ ‘O’ to ‘Ø’
compose ‘/’ ‘o’ to ‘ø’
compose ‘`’ ‘U’ to ‘Ù’
compose ‘`’ ‘u’ to ‘ù’
compose ‘\’’ ‘U’ to ‘Ú’
compose ‘\’’ ‘u’ to ‘ú’
compose ‘^’ ‘U’ to ‘Û’
compose ‘^’ ‘u’ to ‘û’
compose ‘”’ ‘U’ to ‘Ü’
compose ‘”’ ‘u’ to ‘ü’
compose ‘\’’ ‘Y’ to ‘Y’
compose ‘\’’ ‘y’ to ‘y’
compose ‘T’ ‘H’ to ‘P’
compose ‘t’ ‘h’ to ‘p’
compose ‘s’ ‘s’ to ‘ß’
```

```
compose ‘″’ ‘y’ to ‘ÿ’
compose ‘s’ ‘z’ to ‘ß’
compose ‘i’ ‘j’ to ‘ÿ’
```